conference

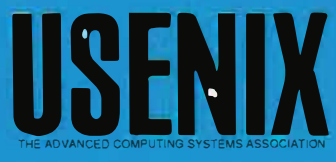
proceedings

# FREENIX Track

## 2004 USENIX Annual Technical Conference

*Boston, MA, USA*
*June 27–July 2, 2004*

**Sponsored by**
**The USENIX Association**




USENIX

FREENIX Track: 2004 USENIX Annual Technical Conference

*Boston, MA, USA, June 27–July 2, 2004*

For additional copies of these proceedings contact:

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA
Phone: 510 528 8649
FAX: 510 548 5738
Email: office@usenix.org
URL: http://www.usenix.org

The price is $30 for members and $40 for nonmembers.
Outside the U.S.A. and Canada, please add
$18 per copy for postage (via air printed matter).

## Past FREENIX Proceedings

| | | |
|---|---|---|
| FREENIX '03 | 2003 | San Antonio, TX $22/30 |
| FREENIX '02 | 2002 | Monterey, CA $22/30 |
| FREENIX '01 | 2001 | Boston, MA $22/30 |
| FREENIX '00 | 2000 | San Diego, CA $22/30 |
| FREENIX '99 | 1999 | Monterey, CA $22/30 |



ISBN 1-931971-22-6

USENIX Association

# Proceedings of the FREENIX Track

## 2004 USENIX Annual Technical Conference

June 27–July 2, 2004
Boston, MA, USA

# Program Organizers

### FREENIX Program Chair

Bart Massey, *Portland State University*
Keith Packard, *Hewlett-Packard Cambridge Research Lab*

### FREENIX Program Committee

Crispin Cowan, *Immunix*
Ralf Nolden, *RWTH Technical University of Aachen*
Jonathan Shapiro, *Johns Hopkins University*
Luis Villa, *Ximian*
Carl Worth, *USC Information Sciences Institute*
Victor Yodaiken, *FSMLabs*

### UseBSD SIG Session Chair

Murray Stokely, *FreeBSD Mall, Inc.*

### UseLinux SIG Session Chair

Theodore Ts'o, *IBM*

### UseLinux Program Committee

Jerry Feldman, *Compaq Computer Corp.*
Bdale Garbee, *HP Linux CTO/Debian*
Jim Gleason, *VA Software*
Jon "maddog" Hall, *Linux International*
Don Marti, *Linux Journal*
Stacey Quandt, *OSDL*

### Extreme Linux SIG Session Chair

Jon "maddog" Hall, *Linux International*

### Extreme Linux Program Committee

Peter Beckman, *Argonne National Laboratory*
Steven A. DuChene *Consultant*
Dan Ferber *SGI*
Michael Fitzmaurice, *Northrup/Grummund*
Forrest Hoffman, *Oak Ridge National Laboratory*
Matt L. Leininger, *Sandia National Laboratory*
Alain Roy, *University of Wisconsin, Madison*
Jennifer Schopf, *Argonne National Laboratory*
Mitchel Williams, *Sandia National Laboratory*

### The USENIX Association Staff

# FREENIX External Reviewers

Robert Bauer
Jim Binkley
Gregory Neil Shapiro

# 2004 USENIX Annual Technical Conference
# FREENIX Track
# June 27–July 2, 2004
# Boston, MA, USA

## Friday, July 2, 2004

### Software Engineering



### System Building

## SIG Sessions

### UseBSD SIG



### UseLinux SIG



### Extreme Linux SIG

# Index of Authors

# Message from Your Friendly FREENIX Program Chairs

We are sure it will be a shock to those who know us to find out that we are composing this message the day it is to be sent off to the printer. Well, OK, maybe not such a shock. Being conference chairs has turned out to be the kind of detailed, exacting, responsible work for which we are entirely ill suited. The good news is that almost nobody will waste precious minutes of their life reading obscure comments in the USENIX proceedings, so our little secret is still safe. Please don't tell.

In spite of ourselves, we think we've put together a pretty nice FREENIX '04 program for you. Of course, the "we" here is collective. The FREENIX Program Committee and the USENIX directors and staff did most of the hard work on our end. Their guidance and tolerance is gratefully acknowledged and appreciated.

Unlike in past years, we decided to hold the program committee meeting in a non-frozen location. Although we were unable to get USENIX to fly everyone to Vanuatu, we did manage to bring the committee to Portland, Oregon. A small improvement from January in Michigan or New Jersey (at least the precipitation was liquid). For two days, we had a chance to review the submissions as a team and select the most suitable papers and abstracts for the program. Most of the accepted submissions were short abstracts which were then assigned to committee members for shepherding into full papers.

The material submitted for FREENIX seems to get better every year, and this year was no exception. Of the 61 submissions we received, we were only able to accept 15. We're sorry to say that there were an especially large number of high-quality submissions that could not be accepted for lack of space in the program. This is always disappointing, but is evidence of the high caliber of work being performed in the F/OSS community and submitted to the USENIX ATC these days.

Of course, a big part of the USENIX experience is a chance to come meet with friends and colleagues. We hope you enjoy the conference this year and we look forward to seeing you.

**Bart Massey and Keith Packard**
**FREENIX Program Chairs**

# Migrating an MVS Mainframe Application to a PC

Glenn S. Fowler, Andrew G. Hume, David G. Korn and Kiem-Phong Vo
AT&T Labs – Research
180 Park Avenue, Florham Park, NJ 07932, USA
{gsf,andrew,dgk,kpv}@research.att.com

## Abstract

Due to advances in computer architecture, performance of the PC now exceeds that of the typical mainframe. However, computing cost on a mainframe continues to greatly exceed that on a PC. Thus, migrating mainframe applications to PC can result in substantial savings. The major stumbling block in doing this is the cost of software migration itself. This paper discusses an experiment in using a software tool approach to migrate a large billing application from MVS running on a mainframe to UNIX running on a PC. We developed tools to port the application from mainframe to PC with minimal code rewriting and enable transferring data cheaply so that processing can be done without interrupting other ongoing mainframe operations. We were able to transfer data from the MVS mainframe to a Linux PC and complete its processing in less total time than if done entirely on the original mainframe.

## 1 Introduction

Approximately 25 years ago John Linderman at Bell Laboratories wrote a Technical Memorandum describing the UNIX *sort* program. The state of Unix computers was such that John concluded that it was faster to sort a file by writing it to tape, transferring the tape to the mainframe to sort, and then loading the result back to the Unix machine. Much has changed in 25 years. Processor speed, memory, disk capacity, and network speeds have followed Moores law and improved exponentially. A 3GHz processor is cheaper now than a 1Mhz processor was then. A gigabyte of memory costs around $300 compared to $30,000 for a megabyte. A 250 gigabyte disk now costs around $250 compared to about $10,000 for a 100 megabyte drive. Meanwhile, networking speeds of 64 kilobits a second have increased to a Gigabit a second. The price/performance of mainframes has lagged behind the PC's to the extent that a mainframe computer with the equivalent power as a PC now costs at least two orders of magnitude higher.

Beyond hardware costs, the software costs for mainframes are also at least an order of magnitude higher than that for PCs. There are two reasons for this. First, the number of mainframes is substantially less than the number of small computers so that software production costs are amortized against a smaller population. Second and more importantly, software development on mainframes now requires specialized, vanishing skills due to the use of antiquated programming languages and tools such as COBOL and JCL. To contain the high cost of software development on mainframes, Information Technology organizations have experimented with various approaches to migrate computing to smaller computers.

The most direct approach to migration is to simply rewrite mainframe applications to run on PCs. This task is difficult as everything from the operating system, databases, the languages, and even the character set representations are different between these computing environments. Further, the people who wrote the original system may no longer be around and there may not be enough documentation to provide a complete description to duplicate. This means that massive code reengineering effort is necessary. Lastly, even if a large software application is successfully rewritten, it may still need to interoperate with other mainframe systems. This means that data migration and/or transcoding must be addressed upfront along with code rewriting and scalable data transport between machines must be available. Although many software tools and techniques for software and data reengineering [2, 6, 7, 15] are available, the risk in conducting such a migration effort is enormous. The peril of this approach was highlighted recently in an article on the New York Times * on the cost overrun in an effort to modernize the Internal Revenue Service software system.

A more reasonable approach adopted by a number of IT organizations is to encapsulate the mainframe by providing standard access to data, and then using PC's to add new functionality. While this strategy does make it possible to add new features more cheaply, the large central costs of maintaining the mainframe remain. In addi-

*http://www.nytimes.com/2003/12/11/business/11irs.html

tion, as the data streams between the different environments diverge, it becomes difficult to perform decision supporting tasks such as data mining that may require fine-grained data integration and manipulation.

The strategy which we chose to explore is to develop tools that allow much of the current mainframe software to run on the PC with little or no change. The original code can be either automatically converted or interpreted. As the software must handle both mainframe and native PC data, tools are provided to transparently and cheaply move data between the two environments. Of course, the software on the new system still requires much of the same skill set to maintain as it did on the mainframe. However, once on the PC, the software can be incrementally rewritten. In this way, significant savings can be realized early with small upfront costs. There have been a few efforts along this direction, most notably the works by Henault [12], Rossen [17] and Townsends [18]. Both of these stopped at translating the Job Control Language scripts into Unix shell scripts.

The rest of the paper describes the tools and techniques developed in an experiment to move a mainframe application to PC's. The ported application was run alongside its mainframe original to test correctness and measure performance.

## 2 The application to be migrated

Daily operations in a corporation like AT&T are dominated by large mainframe applications. It is important for the success of any migration project to quickly show advantage over existing practice. This implies the below criteria for selecting a test case for our approach to software migration:

- *Ease of implementation*: For fast demonstration of concepts, we wanted an application that would showcase the use of tools. In particular, our AST Toolkit [9, 10] provided a large collection of software tools for various aspects of computing, ranging from compression [21] and sorting to scripting [13] and software configuration [8]. So we wanted an application making extensive use of these techniques.

- *Correctness and significance*: To show effectiveness, we wanted an application with sufficient processing complexity, dealing with large data and independent from any MVS database. The database independence aspect enabled running the migrated version on the PC alongside its mainframe counterpart and testing the results. At the same time, cost savings could be shown by measuring the performance of both versions.

The application that we chose was a part of the VTNS Billing Edge biller. This system processed billing records for approximately 340 business customers per month. Billing records were fixed size, 650 bytes, but the number of records per customer could vary anywhere between one and around 40 million. Data processing was done in two cycles. The larger cycle on the $10^{th}$ processed about 290 customers with total data about 560 gigabytes and required about 60 CPU hours of mainframe computing. The smaller cycle on the $31^{st}$ handled the rest of the customers with approximately 140 gigabytes of data and required about 24 CPU hours.

Table 1: Code sizes

| Language | #Files | #LOCs |
|---|---|---|
| COBOL | 30 | 15K |
| JCL | 900 | 100K |

Table 1 summarizes the size of the application. COBOL is the main programming language used on mainframes. JCL is the Job Control Language used to write scripts to execute processes similarly to the shell language [13] on a Unix system. We note that the chosen application is just a small part of the entire biller which is more than 1.5M COBOL LOCs.

Figure 1 shows the data processing for this application. The names of the components are as defined by the overall biller. The workflow among the components is as follows:

- The input data to our application is generated by three external systems encapsulated in the diamond boxes.

- The JCL scripts, `VMURH1` and `VMURH4` (in ellipses), run COBOL programs to process this data into records sorted in various ways. Certain non-key fields of records that compare equal may be summarized to produce various sorted output files. Note also that `VMURH1` is generic and only driven by customer-specific data but `VMURH4` is a JCL script generated per customer.

- The output of `VMURH1` for all customers are then merged together by JCL scripts `VMURH31Q` and `VMURH31!Q` into files by types.

- Finally, the files grouped by types via `VMURH31Q` are processed by a set of ten processes `VTUDHR[01-10]` into files to be processed by other systems.

The internal working of the above software components are opaque to us. However, understanding them is

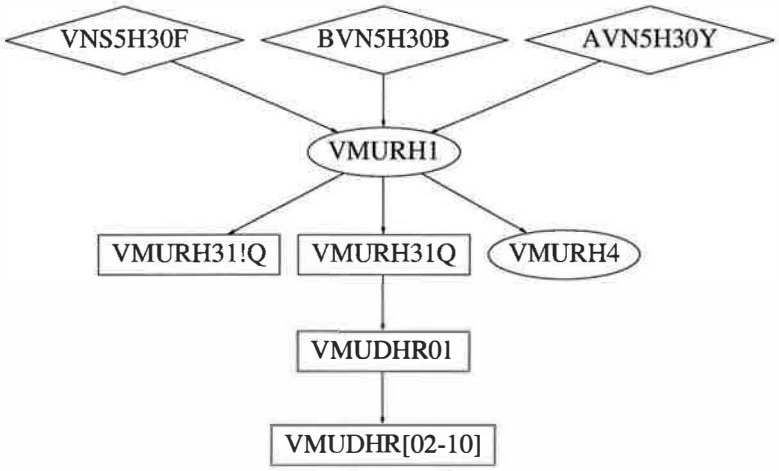



Figure 1: Processing customer data.

not necessary in a tool approach to migration. We only need to ensure that the processes can be compiled and executed or interpreted on the PC's to produce the same data as on the mainframe.

# 3 Software tools

Our AST Toolkit consists of an extensive collection of tools portable across nearly all flavors of Unix. In fact, these tools run transparently on both the PC and the UNIX System Services [1] part of MVS. However, there are major differences between mainframe MVS and PC UNIX that necessitate a number of new tools and techniques:

- Software to copy files between mainframe and PC.
- A COBOL compiler.
- A way to read and execute the MVS job control language, JCL.
- A sort program compatible with the IBM sort program.
- A way to schedule jobs on the PC's to efficiently process data.

## 3.1 Copying data between mainframe and PC

There are number of issues in moving data between MVS and PC. We discuss each below:

- *File system differences:* Much of MVS data are stored in MVS partitioned data sets. MVS partitioned data sets are similar to UNIX archives which can be mapped into Unix directories on the PC. Fortunately, the MVS Unix System Services provides a *cp* command that can extract the individual files from a partitioned data set and copy them from the MVS file system to the UNIX file system. In particular, it copies a partitioned dataset into a Unix directory so that each member becomes a file. Then, the *pax* utility can be used to archive this directory into either *cpio* or *tar* format and transfer the data to the PC via *ftp*.

  COBOL source files, however, are not stored in a partitioned data sets. Instead, they were stored in a sequential data set in a format understood by an MVS tool named *ca-librarian* [3]. As the interface to *ca-librarian* is interactive and menu-based, it is difficult to extract and copy multiple files in bulk. We reengineered this data format and built it into our AST *pax* command. This allows us to simply copy the *ca-librarian* sequential data set and read it on the PC using *pax*.

- *Expensive data movement:* A major problem with moving data between the mainframe and the PC is volume. With existing data connection, transfer rate is between 1 and 2 megabytes per second. A month worth of data for our application is about 700 gigabytes. This means that transferring data from mainframe to PC alone could be anywhere between 97 hours up to 184 hours. Thus, we investigated compression tools such as *gzip* and *compress* to reduce the data size. As observed elsewhere [4, 5, 19], fixed-length records data were amenable to compression by first transforming the data to be column-major instead of row-major. We wrote a simple compressor that transposed rows and columns, then applied run length and static Huffman encoding.

This technique compressed three times better than *compress* and twice better than *gzip* and ran about 4 times faster than both tools. On MVS, we were able to compress a gigabyte of data in 80 seconds and got about a factor of 25 compression. In this way, the 700 gigabytes per month of data could be compressed and transferred in around 20 hours.

- *Different character sets:* MVS uses EBCDIC to store text while Unix uses ASCII. This means that text files must be converted to ASCII en route to the PC and back to EBCDIC in the other direction. Since our compressor was built into the Vcodex package [21] which supports general data transformation, it was a simple matter to add character set transcoding before encoding or after decoding.

## 3.2 COBOL

With about 15 KLOCs of COBOL to be migrated, even if we were proficient in COBOL (and we were not), it would have taken a significant amount of time to rewrite the application in C. Thus, we looked into obtaining a COBOL compiler for PCs. After failing to get proper licensing terms for a commercial COBOL compiler, we started experimenting with openCOBOL, an Open Source COBOL compiler written by Keisuke Nishida [16]. Although this compiler did provide most needed features, there were a number of changes needed to support a large application like ours. We discuss these next:

- *Language additions:* The openCOBOL language lacked a number of features supported by the IBM COBOL compiler. For example, IBM COBOL provides an `ENTRY` statement that enables multiple entries to a procedure and a `WHEN-COMPILED` preset variable stores that date and time that the program is compiled. These features had to be added in order to compile the existing COBOL code.

  The IBM compiler allowed specifications such as `ASSIGN TO DA-S-VMUR102B` without quoting whereas the openCOBOL one required that the dataset name be quoted. We modified the compiler to accept this syntax and to skip over the `DA-S-` prefix and then use the `VMUR102B` as the name of an environment variable to search for to find the actual file name.

  With JCL, multiple files can be specified for each dataset. In that case, the files are virtually concatenated into a single input data stream. We modified the openCOBOL compiler to accept a space separated list of file names as the value of an environment variable and open them sequentially as if they formed a single concatenated file.

  The generation of the `main` program was modified so that a `USING` clause would get the data from the first argument passed to the program.

- *Performance:* Initial testing of the compiler revealed that it could not handle files larger than 2 gigabytes. After fixing this problem, we found the compiler to be too slow mostly due to its arithmetic processing. We were able to rewrite this part of the code and some other parts to improve the speed of compiled code by a factor of five. Adjusting for the relative speed of computation on MVS, the compiler now produces faster code than that produced by the IBM compiler.

- *Character sets*: Since the compiler is ASCII-based, string data in data files must be in ASCII. We started by converting string fields in each record from EBCDIC into ASCII and left numeric fields alone. This was not sufficient since these data files may have string and numeric data in the same columns across different records. The solution was to convert all bytes to ASCII and modify the arithmetic conversion routines of the compiler to convert back to EBCDIC when doing the conversions.

- *Processing compressed data*: We modified the compiler to automatically handle data compressed by the mentioned compressor. By convention, such compressed data are kept in files whose names end in `.qz`. Thus, if the compiler opens such a file for sequential reading, data is automatically decompressed before reading. Conversely, for sequential writing, the data is automatically compressed.

We were able to work with Nishida to add all of the needed language features as well as other modifications into the openCOBOL compiler. In this way, future projects migrating COBOL programs can build on our work.

Finally, the compiler converts each COBOL module to C and then invokes the GNU C compiler, *gcc*, to build an object file. The object files are then linked with a runtime library supporting various COBOL features to make an executable program. We wrote *nmake* makefiles [8] to automate this process. However, given the large number of makefiles that must be written in a large application, we are investigating automatically generating them.

## 3.3 Sorting

The MVS *sort* program has a number of features beyond normal sorting. For example as records are read, files can be created by selecting certain subset of the fields to create new records. Records comparing equal by keys may

also be merged by summing certain field values. The description of what keys to sort on, how to do merging, and what additional files to create is defined in a separate specification called *sort control cards*. These features of MVS *sort* were used extensively in our chosen application.

The AST *sort* program is a superset of the UNIX *sort* utility defined in the POSIX standard. A feature of the AST *sort* not apparent at the command level is that it is just a driver on top of a sorting library designed in the *Disciplines and Methods* paradigm [20]. This paradigm provides a standard API via the *discipline* mechanism to extend library functionality. We were able to duplicate the required functionality of MVS *sort* by writing a few disciplines that make use of MVS sort control cards to process a record when it is read or to merge records compared equal in the same way that MVS *sort* does it. The *sort* disciplines are implemented as shared library plugins. This means that discipline-specific overhead is only incurred after the plugin is loaded at runtime.

For full MVS compatibility, we extended AST *sort* to deal with fixed length records and binary coded decimal fields. Similar to modifications to the COBOL compiler, the sort program could also handle concatenation of files and data compression with the `.qz` suffix.

On MVS, our *sort* runs about 5% faster than MVS *sort*. However, the two *sort* programs occasionally produces records in different order since ours is stable while MVS *sort* is not. That is, our *sort* preserves file order for records that compare equal by keys while MVS *sort* does not provide this guarantee.

## 3.4 JCL

JCL, the job control language for MVS, plays much the same role as the UNIX shell does in that it invokes programs or scripts in some order and takes actions based on the results. The chosen application executes over 100 thousand lines of JCL about 90% of which are generated and the remaining 10% are fixed. A JCL script is generated for each customer by accessing the DB2 database. These scripts merely call MVS *sort* with various control card decks generated from within the scripts. To eliminate these JCL scripts would require access to the database from UNIX. To avoid this complexity, we kept the generation of these scripts on MVS and then processed them on UNIX.

We wrote *jcl*, a JCL interpreter, that allows the use of the hierarchical Unix file system instead of the flat MVS file names via file name prefix mapping. For example, a partitioned data set for control cards, say `SYS1.CTLCDLIB`, would be mapped to the directory `${BILLROOT}/cntlcard` so that its control cards would be stored in separate files in this directory. *jcl* first parses JCL scripts into a linked list of program step structures. This list is traversed to either generate *ksh* shell scripts or to execute. *jcl* also provides debugging support that can be used to determine the overall structure and relationships between a collection of JCL scripts. For example, the `--noexec` option interprets the JCL but does not execute external programs and the `--list=`*item* option lists the *items* referenced by each JCL step.

## 3.5 Job scheduling

An application on MVS consists of a number of jobs some of which can run in parallel while others must wait until some other set of jobs finish before they can start. Let $A$ and $B$ be two jobs. We say that there is a directed edge $A \rightarrow B$ if there is a constraint that $A$ must be completed before $B$ can start. In this way, the set of jobs and constraints form a directed graph called the *scheduling graph*.

Figure 2 shows a slice of the scheduling graph for our application based on the processing of just two customers *xx* and *yy*. This essentially executes the workflow presented in Figure 1 except that all customers are now being considered together so there are more opportunities for parallelization. For example, as soon as the data produced by the external systems AVN5H30Y, VNS5H30F and BVN5H30B for a particular customer are available, the VMURH1 process for that customer can be started. As long as there are enough processing power, the scheduler starts many such processes in parallel. The results from these processes are further processed. In particular, the merged results by VMURH31Q are passed on to the VTUDHR processes.

By necessity, a scheduling graph must be acyclic so that the jobs can be scheduled. In general, jobs may have attributes associated with them such as completion time or memory and disk resource constraints. In that case, it is desirable to compute a schedule that optimizes some parameters based on these attributes. When only a single processor is available, any topological sort ordering of the jobs produces a valid optimal schedule. However, on a system with multiple processors, the scheduling problem is known to be NP-hard[11].

The MVS scheduler, New-Dimension, allows scheduling constraints to be specified by filling out form tables. Then, the scheduler controls resources of the system and sequences the jobs appropriately. For the PC's, we opted to write a simple scheduler, which reads lines from one or more queues specified as files, and runs a command for each line it reads. At startup, the scheduler is given a list of process resources and will run a single job at a time through each resource. If all resources are in use, the scheduler blocks until one becomes available. If all the file queues are empty, the

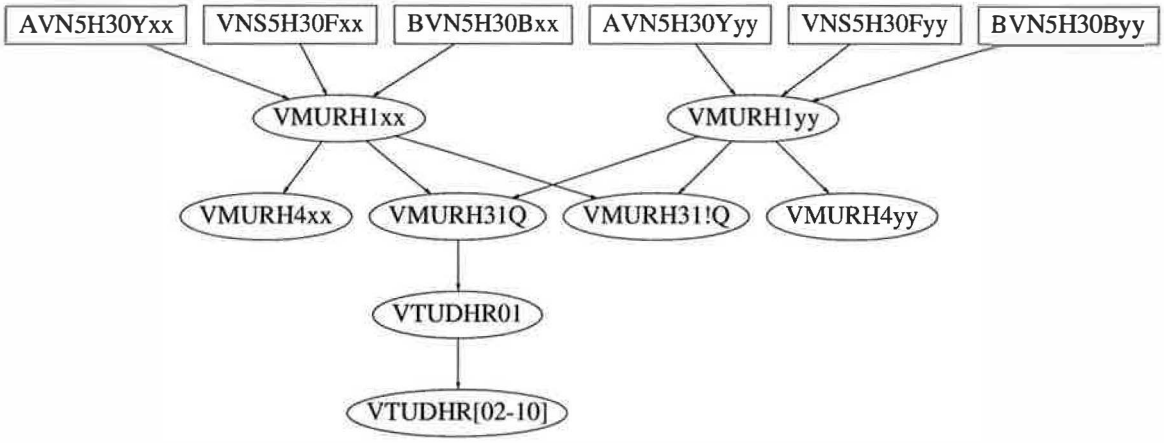



Figure 2: A scheduling graph.

scheduler will block until one of the file queues contains some input to process.

The simple scheduler is adequate for our prototype. However, in general, MVS scheduling has many facets that we did not account for. We are investigating writing a tool to read the MVS scheduling tables and perform the appropriate scheduling on the PC's.

# 4 Selecting a platform

We wanted to select a platform of hardware and software fast enough to do the processing, able to store the data, able to run all of the software, and at as low cost as possible. Based on price performance considerations, we built a cluster of two machines, each with a 2.8 MHz. Intel Pentium 4 processor, 1 gigabyte of 400 MHZ DDR RAM, and two 256 gigabyte SATA disk drives. The machines are networked with Gigabit Ethernet. Unit testing various CPU intensive programs showed that these machines were about 7 times as fast as the MVS systems. The SATA disk drives were able to tranfer about 100 megabytes a second. The entire cost for both machines was under $4,000.

Since most of the processing was per customer and could be done in parallel, we kept the machines loosely coupled with separate file systems instead of using a single shared file system. As data processing was intensive, avoiding the overhead of a shared file system such as NFS was a win overall. In data processing phases where files must be merged, they could be copied to wherever needed. Keeping a loosely coupled cluster of machines made it simple to disconnect a bad machine or to add new machines as needed. This increased fault-tolerance and scalability.

The choice of an operating system was constrained by the software to be run. The use of the AST Toolkit did not constrain this choice in any way. However, the openCOBOL compiler required the GNU-C compiler *gcc* and certain GNU libraries. We finally chose Redhat Linux 9.0 primarily because we felt it would be easier to integrate our solution into the billing application supported by IBM. Our concerns with Redhat Linux were primarily its erratic I/O performance. For example, the throughput was actually higher by running a single customer at a time on each system rather than running multiple customers in parallel. However, we were not locked into any operating system. We experimented running the software with FreeBSD Unix. In contrast to Linux, the throughput did increase when running two customers in parallel on FreeBSD Unix. Finally, a UNIX based solution was chosen rather than Windows mostly for reliability consideration. However, all the tools could be run on Windows using the UWIN software [14].

# 5 Results

Table 2: October 10, 2003 cycle.

| Data | #Files | Raw | Comp. |
|---|---|---|---|
| From mainframe | 300 | 560GB | 22GB |
| Generated on PC | 1160 | 280GB | 17GB |

For testing, we processed data for the October 10, 2003 cycle. Table 2 summarizes the data that were transferred from the mainframe and generated on the PC's. Theoretically, the data could be compressed on the mainframe in about 12.5 hours and transferred to the PC's in about 6 hours for a total of 18.5 hours. However, it took us over 24 hours to move the data from the mainframe to the PC's, mostly due to long waiting time for mainframe

tape drives. Once the data arrived on the PC's, processing completed in under 19 hours. Thus, even with the slow data transferring time, the output data was generated in about 43 hours. The same job took a total of 60 hour processing time on the mainframe. Note that, in this case, we waited until all files were copied to the PC's before processing. Our scheduler could be modified to allow overlapping of data transferring and data processing to reduce elapse time even further. In addition, our set up could be easily scaled up by adding more PC's. We estimated that with a 4 processor system, we could process this data in under 12 hours.

CPU cycles on the mainframe costs about $20/hour and network charge for data transmission to and from mainframe is around $5/Gbyte. Thus, if the above data is representative of the $10^{th}$ cycle, it could be compressed and transferred between the mainframe and PC's for less than $1000. Generously doubling this to cover both the $10^{th}$ and the end of the month cycles, the total cost for data transfer each month would be less than $2000. Assuming that the PC's cost $2000/month to own and operate, the total cost for migrating to the PC's would be under $4000/month. As the current cost to run the application on the MVS mainframe is estimated to be about $20000/month, the data processing cost can be reduced by more than a factor of 5 for this application using our approach.

# 6 Conclusions

We presented a methodology to migrate mainframe applications to PC's based on software tools. This approach minimized software and data reengineering and enabled smooth transition between the mainframe and the PC's. The work took place over a six month period and was carried out by a small team of software experts without prior MVS or COBOL skills. Much of the effort was spent in learning MVS, COBOL, and in writing reusable tools. Thus, the work could be easily duplicated on more ambitious problems with far bigger payoffs.

In an experiment using the developed tools to move a small but significant mainframe application to a system of two PC's costing less than $4000, we showed that over 80% of the monthly computing cost could be reduced, saving more than $16000 per month. This cost improvement was conservative. Most of the cost was driven by moving data from the mainframe to the PC's and that could be easily eliminated by getting the data directly to the PC's. The migration of the entire billing application to PC's might save up to 90% of the ongoing costs.

Beyond software migration, the work on compression and sorting were of a general nature. For example, the table compressor could be used to compress any database tables using fixed length records. It was shown elsewhere [19] that mainframe data of the type mentioned here could be compressed by factors anywhere between 50 to 100 to 1. Thus, the compression tool could be used to save disk space and tape usage on the mainframe. Our *sort* tool employs better sorting algorithms than MVS *sort* and the commercial *Syncsort* (http://www.syncsort.com/) software used in original project. Thus, it can be used to both improve processing and save the rather steep licensing fees being paid yearly.

Looking forward, there are a number of threads to be developed. For example, our test case does not involve database and report generation issues which are important in large applications. The questions of reliability, effective scheduling of processes, and optimal partitioning and placement of large data on a cluster of loosely coupled PC's are of independent interest. Such problems should be dealt with in conjunction with migrating a larger and more comprehensive application. We are looking into that.

# Acknowledgements

We would like to thank a number of colleagues who provided assistance in this project. Silvia Coble and Robert Cummins suggested the application to migrate and helped obtaining data to understand what was needed. Jim LaBarge from IBM helped with MVS accounts and the Unix System Services. Doug Blewett from AT&T Research helped select the hardware platform, assembled it, and got it running. Finally, Chris Olsen administered our system.

# References

[1] z/OS UNIX System Services. In *http://www.ibm.com/servers/eserver/zseries/zos/unix/*.

[2] P. Aiken. *Data Reverse Engineering: Slaying the Legacy Dragon*. McGraw-Hill, 1995.

[3] Computer Associates. CA-Librarian User Guide 4.3 for OS/390, z/OS, and VSE. In *http://www.dcs.rochester.edu/Documentation/CA-Librarian%20V4.3/lib43%20User%20Guide.pdf*, 2001.

[4] A. Buchsbaum, D. Caldwell, K. Church, G.S. Fowler, and S. Muthukrishnan. Engineering the Compression of Massive Tables: An Experimental Approach. *Proc. 11th ACM-SIAM Symp. on Disc. Alg.*, pages 175–184, 2000.

[5] A. Buchsbaum, G.S. Fowler, and R. Giancarlo. Improving Table Compression with Combinatorial Optimization. *Proc. 13th ACM-SIAM Symp. on Disc. Alg.*, pages 213–222, 2002.

[6] Y.-F. Chen. The C Program Database and Its Applications. In *Proc. of the Summer 1989 USENIX Conference*, pages 157–171, June 1989.

[7] Y.-F. Chen, G.S. Fowler, E. Koutsofios, and R.S. Wallach. Ciao: A Graphical Navigator for Software and Document Repositories. In *International Conference on Software Maintenance*, 1995.

[8] Glenn S. Fowler. The Fourth Generation Make. In *Proc. of the USENIX 1985 Summer Conference*, June 1985.

[9] Glenn S. Fowler, David G. Korn, and Kiem-Phong Vo. Principles for Writing Reusable Libraries. In *Proc. of the ACM SIGSOFT Symposium on Software Reusability*, pages 150--160. ACM Press, 1995.

[10] G.S. Fowler, D.G. Korn, S.C. North, and K.-P. Vo. The AT&T AST OpenSource Software Collection. In *Proc. of the Summer 2000 Usenix Conference*. USENIX, 2000.

[11] R.L. Graham, E.L. Lawler, J.K. Lenstra, and A.H.G. Rinnoy Kan. Optimization and approximation in deterministic sequencing ans scheduling: a survey. In *Ann. of Disc. Math.*, pages 5:287–326, 1979.

[12] H. Henault. A.C.M.U. : Automate de Conversion MVS-UNIX. In *http://www.hhns.fr/products/acmu/acmudoc.html*.

[13] David G. Korn. ksh: An Extensible High Level Language. In *Proc. of the Usenix VHLL Conference*, 1994.

[14] David G. Korn. Porting UNIX to Windows NT. In *Proc. of the 1997 Usenix Conference*. USENIX, 1997.

[15] H.A. Muller, J.H. Jahnke, D.B. Smith, M.D. Storey, S.R. Tilley, and K. Wong. Reverse engineering: a roadmap. In *ICSE — Future of SE Track*, pages 47--60, 2000.

[16] K. Nishida. OpenCOBOL Home Page. In *http://www.opencobol.org*, 2003.

[17] E. Rossen. The MVS to UNIX migration HOWTO. In *http://people.linux-gull.ch/rossen/software/migration/migration.html*.

[18] O. Townsends. The Vancouver Utilities for Unix and Linux. In *http://www.uvsoftware.ca*.

[19] B.D. Vo and K.-P. Vo. Using Column Dependencies to Compress Tables. Data Compression Conference, 2004.

[20] Kiem-Phong Vo. The discipline and method architecture for reusable libraries. *Software—Practice and Experience*, 30:107–128, 2000.

[21] Kiem-Phong Vo. Vcodex: A Platform of Data Tranformers. *Work in progress*, 2004.

# C-JDBC: Flexible Database Clustering Middleware

Emmanuel Cecchet
*INRIA Rhône-Alpes*
emmanuel.cecchet@inria.fr

Julie Marguerite
*ObjectWeb Consortium*
julie.marguerite@objectweb.org

Willy Zwaenepoel
*EPF Lausanne*
willy.zwaenepoel@epfl.ch

## Abstract

Large web or e-commerce sites are frequently hosted on clusters. Successful open-source tools exist for clustering the front tiers of such sites (web servers and application servers). No comparable success has been achieved for scaling the backend databases. An expensive SMP machine is required if the database tier becomes the bottleneck. The few tools that exist for clustering databases are often database-specific and/or proprietary.

Clustered JDBC (C-JDBC) addresses this problem. It is a freely available, open-source, flexible and efficient middleware for database clustering. C-JDBC presents a single virtual database to the application through the JDBC interface. It does not require any modification to JDBC-based applications. It furthermore works with any database engine that provides a JDBC driver, without modification to the database engine. The flexible architecture of C-JDBC supports large and complex database cluster architectures offering various performance, fault tolerance and availability tradeoffs.

We present the design and the implementation of C-JDBC, as well as some uses of the system in various scenarios. Finally, performance measurements using a clustered implementation of the TPC-W benchmark show the efficiency and scalability of C-JDBC.

## 1. Introduction

Database scalability and high availability can be achieved in the current state-of-the-art, but only at very high expense. Existing solutions require large SMP machines and high-end RDBMS (Relational Database Management Systems). The cost of these solutions, both in terms of hardware prices and software license fees, makes them available only to large businesses.

Clusters of commodity machines have largely replaced large SMP machines for scientific computing because of their superior price/performance ratio. Clusters are also used to provide both scalability and high availability in data server environments. This approach has been successfully demonstrated, for instance, for web servers and for application servers [9]. Success has been much more limited for databases. Although there has been a large body of research on replicating databases for scalability and availability [3], very few of the proposed techniques have found their way into practice [17].

Recently, commercial solutions such as Oracle Real Application Clusters [16] have started to address cluster architectures using a shared storage system such as a SAN (Storage Area Network). The IBM DB2 Integrated Cluster Environment [5] also uses a shared storage network to achieve both fault tolerance and performance scalability.

Open-source solutions for database clustering have been database-specific. MySQL replication [14] uses a master-slave mechanism with limited support for transactions and scalability. Some experiments have been reported using partial replication in Postgres-R [13]. These extensions to existing database engines often require applications to use additional APIs to benefit from the clustering features. Moreover, these different implementations do not interoperate well with each other.

We present Clustered JDBC (C-JDBC), an open-source middleware solution for database clustering on a shared-nothing architecture built with commodity hardware. C-JDBC hides the complexity of the cluster and offers a single database view to the application. The client application does not need to be modified and transparently accesses a database cluster as if it were a centralized database. C-JDBC works with any RDBMS that provides a JDBC driver. The RDBMS does not need any modification either, nor does it need to provide distributed database functionalities. Load distribution, fault tolerance and failure recovery are all handled by C-JDBC. The architecture is flexible and can be distributed to support large clusters of heterogeneous databases with various degrees of performance, fault tolerance and availability.

With C-JDBC we hope to make database clustering available in a low-cost and powerful manner, thereby spurring its use in research and industry. Although C-JDBC has only been available for a few months, several installations are already using it to support various database clustering applications.

The outline of the rest of this paper is as follows. Section 2 presents the architecture of C-JDBC and the role of each of its components. Section 3 describes how fault tolerance is handled in C-JDBC. Section 4 discusses horizontal and vertical scalability. Section 5 documents some uses of C-JDBC. Section 6 describes measurement results using a clustered implementation of the TPC-W benchmark. We conclude in Section 7.

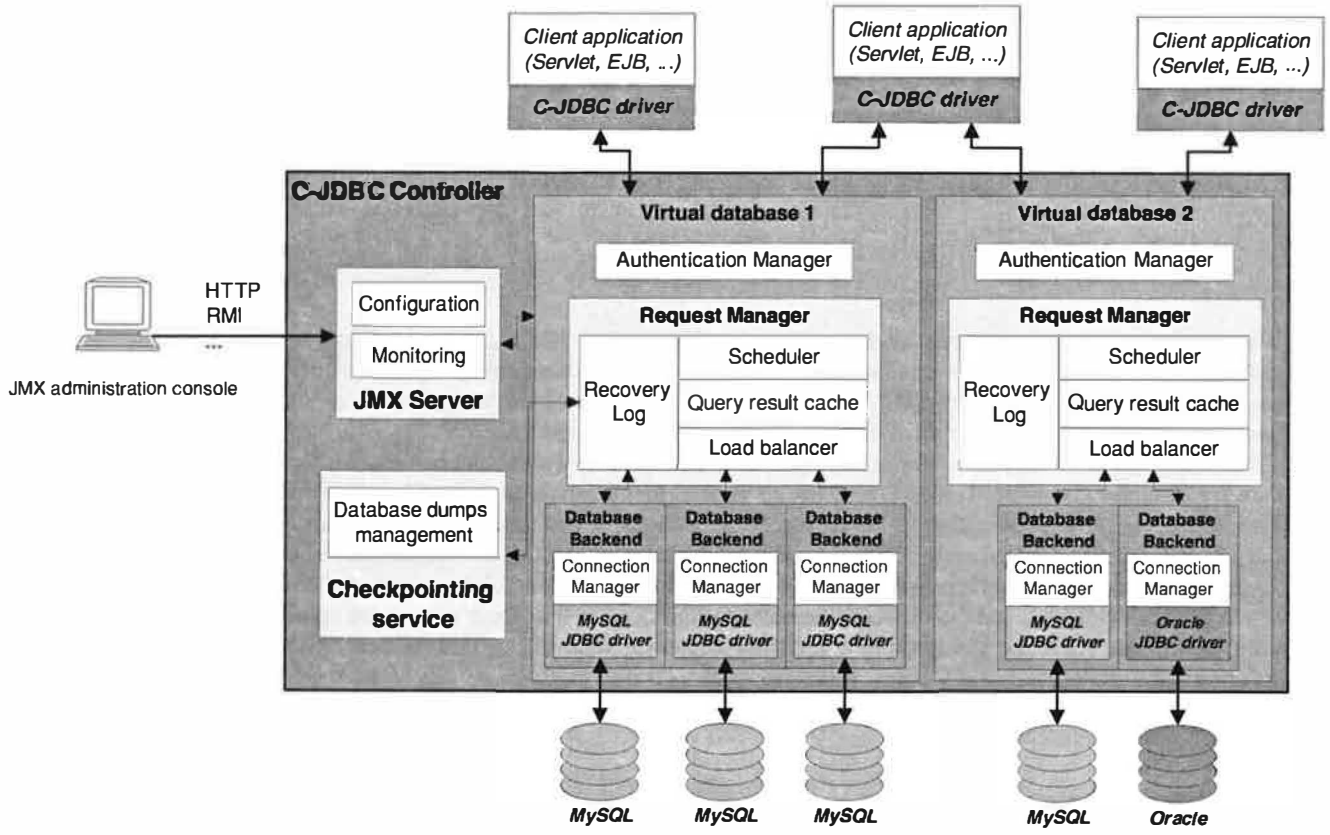



**Figure 1. C-JDBC architecture overview**

## 2. C-JDBC architecture

This section provides a functional overview of C-JDBC, its overall architecture, and its key components.

### 2.1. Functional overview

JDBC™ is a Java API for accessing virtually any kind of tabular data [19]. C-JDBC (Clustered JDBC) is a Java middleware for database clustering based on JDBC. It turns a collection of possibly heterogeneous databases into a single virtual database. No changes are needed to the application or to the databases.

Various data distributions are supported: the data can either be fully replicated, partially replicated, or partitioned, depending on the needs of the application. The degree of replication and the location of the replicas can be specified on a per-table basis. Currently, the tables named in a particular query must all be present on at least one backend. In dynamic content servers, one of the target environments for C-JDBC clusters, this requirement can often be met, since the queries are known ahead of time. Eventually, we plan to add support for distributed execution of a single query.

Routing of queries to the various backends is done automatically by C-JDBC, using a read-one write-all approach. A number of strategies are available for load balancing, with the possibility of overriding these strategies with a user-defined one. C-JDBC can be configured to support query response caching and fault tolerance. Finally, larger and more highly-available systems can be built by suitable combinations of individual C-JDBC instances.

C-JDBC also provides additional services such as monitoring and logging. The controller can be dynamically configured and monitored through JMX (Java Management eXtensions) either programmatically or using an administration console.

### 2.2. Architecture

Figure 1 gives an overview of the different C-JDBC components. The client application uses a C-JDBC driver that replaces the database-specific JDBC driver but offers the same interface. The C-JDBC controller is a Java program that acts as a proxy between the C-JDBC driver and the database backends. The controller exposes a single database view, called a *virtual database*, to the C-JDBC driver and thus to the application. A controller can host multiple virtual databases, as shown in the figure. Each virtual database has its own request manager that defines its request scheduling, caching and load balancing policies.

The database backends are accessed through their native JDBC driver. If the native driver is not capable of connection pooling, C-JDBC can be configured to provide a connection manager for this purpose.

In the rest of this section we discuss the key components of the C-JDBC architecture, in particular the driver and the request manager. Other components that provide standard functionality such as authentication management, connection management and configuration support are not discussed further. More detail on these components and other aspects of C-JDBC can be found at http://c-jdbc.objectweb.org.

### 2.3. C-JDBC driver

The C-JDBC driver is a hybrid type 3 and type 4 JDBC driver[19]. It implements the JDBC 2.0 specification and some extensions of the JDBC 3.0 specification. All processing that can be performed locally is implemented inside the C-JDBC driver. For example, when an SQL statement has been executed on a database backend, the result set is serialized into a C-JDBC driver ResultSet that contains the logic to process the results. Once the ResultSet is sent back to the driver, the client can browse the results locally.

All database-dependent calls are forwarded to the C-JDBC controller that issues them to the database native driver. The native database driver is a type 3 JDBC driver. SQL statement executions are the only calls that are completely forwarded to the backend databases. Most of the C-JDBC driver remote calls can be resolved by the C-JDBC controller itself without going to the databases.

The C-JDBC driver can also transparently fail over between multiple C-JDBC controllers, implementing horizontal scalability (see Section 4).

### 2.4. Request manager

The request manager contains the core functionality of the C-JDBC controller. It is composed of a scheduler, a load balancer and two optional components: a recovery log and a query result cache. Each of these components can be superseded by a user-specified implementation.

#### 2.4.1. Scheduler

When a request arrives from a C-JDBC driver, it is routed to the request manager associated with the virtual database. Begin transaction, commit and abort operations are sent to all backends. Reads are sent to a single backend. Updates are sent to all backends where the affected tables reside. Depending on whether full or partial replication is used (see Section 2.4.3), this may be one, several or all backends. SQL queries containing macros such as RAND() or NOW() are rewritten on-the-fly with a value computed by the scheduler so that each backend stores exactly the same data.

All operations are synchronous with respect to the client. The request manager waits until it has received responses from all backends involved in the operation before it returns a response to the client.

If a backend executing an update, a commit or an abort fails, it is disabled. In particular, C-JDBC does not use a 2-phase commit protocol. Instead, it provides tools to automatically re-integrate failed backends into a virtual database (see Section 3).

At any given time only a single update, commit or abort is in progress on a particular virtual database. Multiple reads from different transactions can be going on at the same time. Updates, commits and aborts are sent to all backends in the same order.

#### 2.4.2. Query result cache

An optional *query result cache* can be used to store the result set associated with each query. The query result cache reduces the request response time as well as the load on the database backends. By default, the cache provides strong consistency. In other words, C-JDBC invalidates cache entries that may contain stale data as a result of an update query. Cache consistency may be relaxed using user-defined rules. The results of queries that can accept stale data can be kept in the cache for a time specified by a staleness limit, even though subsequent update queries may have rendered the cached entry inconsistent.

We have implemented different cache invalidation granularities ranging from database-wide invalidation to table-based or column-based invalidation with various optimizations.

#### 2.4.3. Load balancer

If no cache has been loaded or a cache miss has occurred, the request arrives at the *load balancer*.

C-JDBC offers various load balancers according to the degree of replication the user wants. Full replication is easy to handle. It does not require request parsing since every database backend can handle any query. Database updates, however, need to be sent to all nodes, and performance suffers from the need to broadcast updates when the number of backends increases.

To address this problem, C-JDBC provides partial replication in which the user can define database replication on a per-table basis. Load balancers supporting partial replication must parse the incoming queries and need to know the database schema of each backend. The schema information is dynamically gathered. When a backend is enabled, the appropriate methods are called on the JDBC DatabaseMetaData information of the backend native driver. Database schemas can also be specified statically by way of a configuration file. The schema is updated dynamically on each *create* or *drop* SQL statement to accurately reflect each backend schema.

Among the backends that can treat a read request (all of them with full replication), one is selected according to the load balancing algorithm. Currently implemented algorithms are round robin, weighted round robin and least pending requests first (the request is sent to the node that has the least pending queries).

#### 2.4.4. Optimizations

To improve performance, C-JDBC implements *parallel transactions*, *early response to update, commit, or abort requests*, and *lazy transaction begin*.

With parallel transactions, operations from different transactions can execute at the same time on different backends. Early response to update, commit or abort allows the controller to return the result to the client application as soon as one, a majority or all backends have executed the operation. Returning the result when the first backend completes the command offers the latency of the fastest backend to the application. When early response to update is enabled, C-JDBC makes sure that the order of operations in a single transaction is respected at all backends. Specifically, if a read follows an update in the same transaction, that read is guaranteed to execute after the update has executed.

Finally, with lazy transaction begin, a transaction is started on a particular backend only when that backend needs to execute a query for this transaction. An update query on a fully replicated cluster causes a transaction to be started on all backends. In contrast, for read-only transactions, a transaction is started only on the backend that executes the read queries of the transaction. On a read-mostly workload this optimization significantly reduces the number of transactions that need to be initiated by an individual backend.

## 3. Fault tolerance

C-JDBC provides checkpoints and a recovery log to allow a backend to restart after a failure or to bring new backends into the system.

### 3.1. Checkpointing

A checkpoint of a virtual database can be performed at any point in time. Checkpointing can be manually triggered by the administrator or automated based on temporal rules.

Taking a snapshot of a backend while the system is online requires disabling this backend so that no updates occur on it during the backup. The other backends remain enabled to answer client requests. As the different backends of a virtual database need to remain consistent, backing up a backend while leaving it enabled would require locking all tables in read mode and thus blocking all updates on all backends. This is not possible when dealing with large databases where copying the database content may take hours.

The checkpoint procedure starts by inserting a checkpoint marker in the recovery log (see Section 3.2). Next, the database content is dumped. Then, the updates that occurred during the dump are replayed from the recovery log to the backend, starting at the checkpoint marker. Once all updates have been replayed, the backend is enabled again.

C-JDBC uses an ETL (Extraction Transforming Loading) tool called Octopus [10] to copy data to or from databases. The database (including data and metadata) is stored in a portable format. Octopus re-creates the tables and the indexes using the database-specific types and syntax.

### 3.2. Recovery log

C-JDBC implements a recovery log that records a log entry for each begin, commit, abort and update statement. A log entry consists of the user identification, the transaction identifier, and the SQL statement. The log can be stored in a flat file, but also in a database using JDBC. A fault-tolerant log can then be created by sending the log updates to a virtual C-JDBC database with fault tolerance enabled. Figure 2 shows an example of a fault-tolerant recovery log. The log records are sent to a virtual database inside the same C-JDBC controller as the application database, but this virtual database could have been hosted on a different controller as well. Backends used to store the log can be shared with those used for the application virtual database, or separate backends can be used.

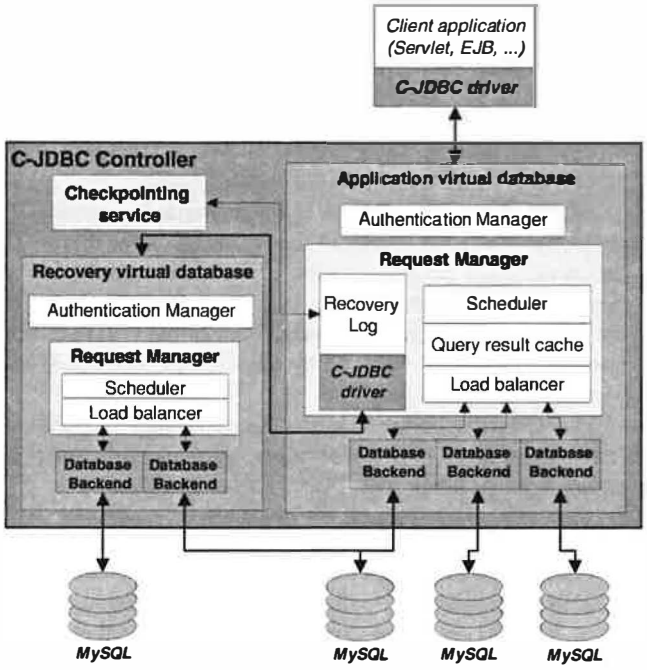



**Figure 2. Fault tolerant recovery log example**

## 4. Horizontal and vertical scalability

The C-JDBC controller is potentially a single point of failure. *Horizontal scalability* reduces the probability of system failure by replicating the C-JDBC controller. To support a large number of database backends, we also provide *vertical scalability* to build a hierarchy of backends.

### 4.1. C-JDBC horizontal scalability

Horizontal scalability prevents the C-JDBC controller from becoming a single point of failure. We use the JGroups [2] group communication library to synchronize the schedulers of the virtual databases that are distributed over several controllers. Figure 3 gives an overview of the C-JDBC controller horizontal scalability.

When a virtual database is loaded in a controller, a group name can be assigned to the virtual database. This group name is used to communicate with other controllers hosting the same virtual database. At initialization time, the controllers exchange their respective backend configurations. If a

controller fails, a remote controller can recover the backends of the failed controller using the information gathered at initialization time.

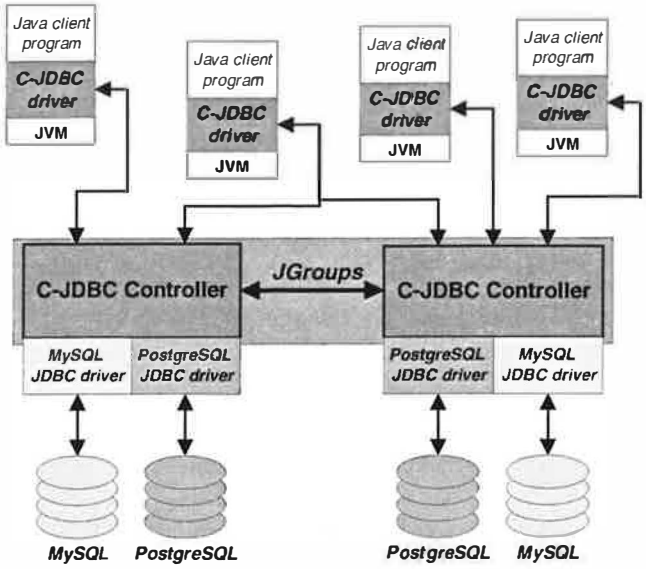



**Figure 3. C-JDBC horizontal scalability**

C-JDBC relies on JGroups' reliable and ordered message delivery to synchronize write requests and demarcate transactions. Only the request managers contain the distribution logic and use group communication. All other C-JDBC components (scheduler, cache, and load balancer) remain the same.

### 4.2. C-JDBC vertical scalability

It is possible to nest C-JDBC controllers by re-injecting the C-JDBC driver into the C-JDBC controller. Figure 4 illustrates an example of a 2-level C-JDBC composition.

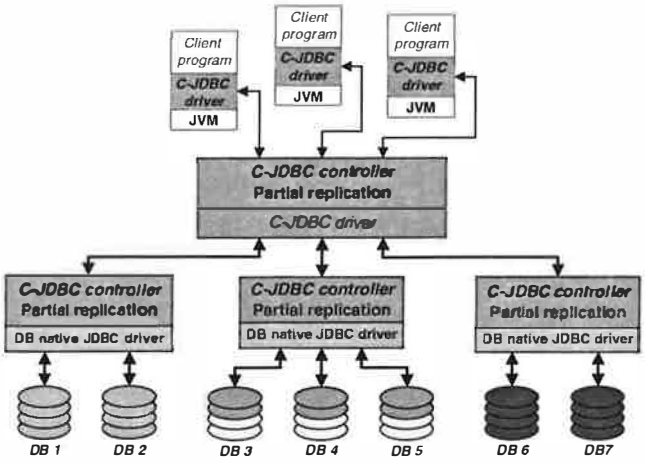



**Figure 4. C-JDBC vertical scalability**

The top-level controller has been configured for partial replication with three database backends that are virtual databases implemented by other C-JDBC controllers. The C-JDBC driver is used as the backend native driver to access the underlying controller. In general, an arbitrary tree structure can be created. The C-JDBC controllers at the different levels are interconnected by C-JDBC drivers. The native database drivers connect the leaves of the controller hierarchy to the real database backends.

Vertical scalability may be necessary to scale an installation to a large number of backends. Limitations in current JVMs restrict the number of outgoing connections from a C-JDBC driver to a few hundreds. Beyond that, performance drops off considerably. Vertical scalability spreads the number of connections over a number of JVMs, retaining good performance.

### 4.3. Mixing horizontal and vertical scalability

To deal with very large configurations where both high availability and high performance are needed, one can combine horizontal and vertical scalability. Figure 5 shows an example of such a configuration. The top-level C-JDBC controllers use horizontal scalability for improved availability. Additional controllers are cascaded to provide performance by way of vertical scalability.

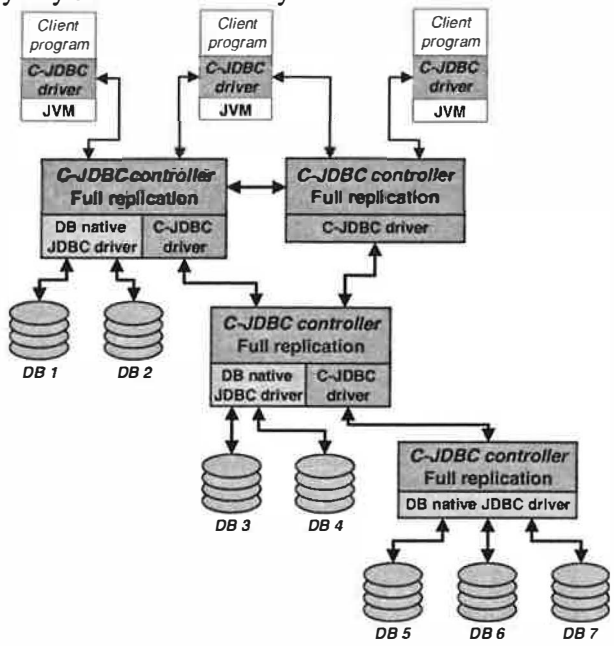


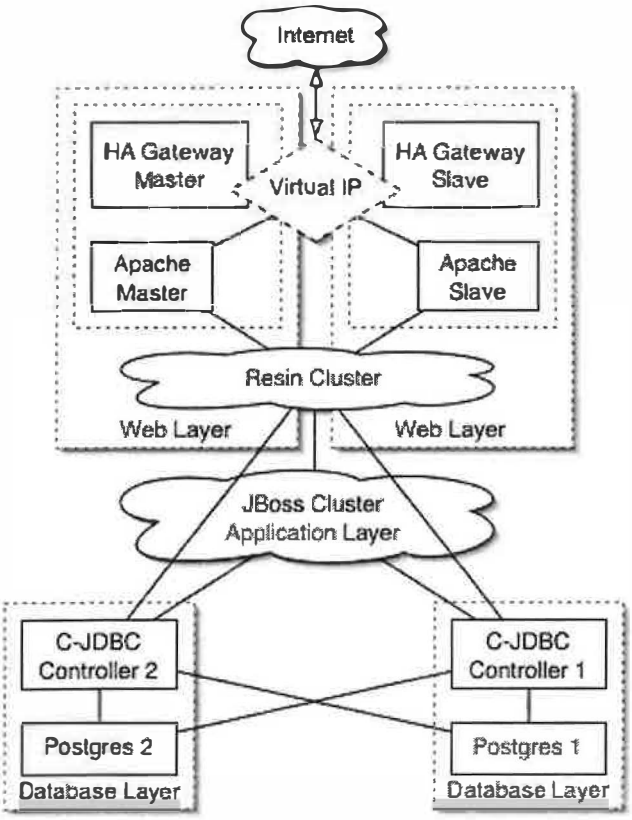

**Figure 5. Cascading C-JDBC controllers**

In this configuration the top-level controllers would normally be configured with early response to updates and commits (see Section 2.4.4) so that the updates can propagate asynchronously down the tree. Read queries are initially sent to the backends DB1 and DB2 that are connected directly to the top-level controller. Once they become loaded, the load balancer in the top-level controller starts to send queries to the lower–level controller, to be executed by backends DB3 and DB4. The more load the system receives, the deeper in the tree the requests go. In this example, the failure of the middle-level controller makes DB5, DB6 and DB7 unavail-

able. To remedy this situation, one could use horizontal scalability to replicate the middle-level controller.

## 5. Sample uses of C-JDBC

Although C-JDBC has only recently been made available, we have already seen considerable use by others. C-JDBC users have different interest and usage scenarios. The flexibility of C-JDBC permits tuning the system for a variety of needs. We present four different use cases focusing on different sets of features. The first example shows how to build a low-cost highly-available system. The next use case demonstrates the combination of portability, performance scalability and high availability for a large production environment with thousands of users and a wide range of operating systems and database backends. The third scenario features a flood alert system where C-JDBC is used to support fast disaster recovery. Finally, the last use case focuses on performance and explains how C-JDBC is used to benchmark J2EE clusters.




Source: http://www.budget-ha.com

**Figure 6. Budget high availability solution from budget-ha.com**

### 5.1. Budget High Availability

Our experience indicates that most C-JDBC users are interested in clustering primarily to provide high availability. Their goal is to eliminate every single point of failure in the system. Budget-HA.com [6] has built a solution from open-source components providing a high-availability infrastructure "on a budget". Figure 6 gives an overview of the proposed 3-tier J2EE infrastructure.

The high availability of the web tier combines Linux-HA with a cluster of Resin servlet containers [7]. The JBoss J2EE server provides clustering features to ensure high availability of the business logic in the application tier. The database tier uses C-JDBC with full replication on two PostgreSQL backends to tolerate the failure of a backend.

The C-JDBC controller is also replicated. Both controllers share the two PostgreSQL backends so that the failure of one controller does not make the system unavailable. In this configuration, the system survives the failure of any component. The minimum hardware configuration requires 2 nodes, each of them hosting an instance of Resin, JBoss, a C-JDBC controller and a PostgreSQL backend.

### 5.2. OpenUSS: University Support System

OpenUSS [15] provides an open-source system for managing computer assisted learning and computer assisted teaching applications. There are currently 11 universities in Europe, Indonesia and Mexico using OpenUSS.

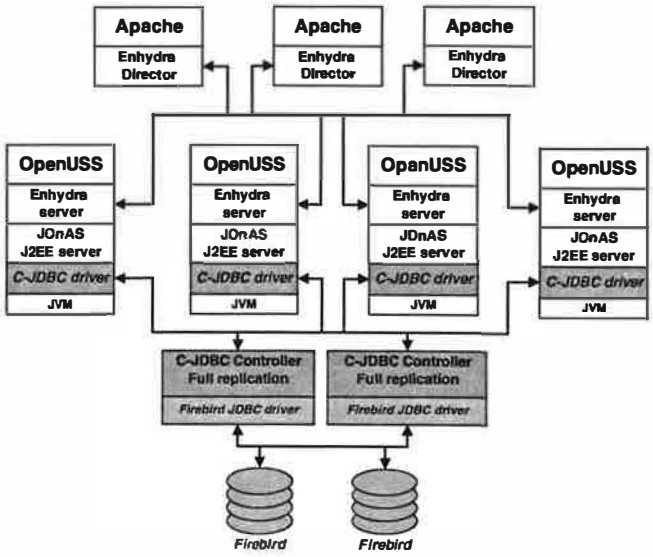



**Figure 7. OpenUSS setup at University of Muenster**

The largest OpenUSS site runs at University of Muenster in Germany. The system manages more than 12,000 students and over 1,000 instructors. The average workload consists of 180,000 to 200,000 accesses per day to web pages that are dynamically generated from data stored in the database. Lecture materials (PDF documents, slides, etc.) are also stored in the database in the form of BLOBs (Binary Large Objects).

To provide both performance scalability and high availability, C-JDBC is used to replicate the Firebird database backends. Figure 7 shows the C-JDBC configuration used at University of Muenster to run OpenUSS.

Three Apache servers function as frontends. Enhydra Director is used as an Apache module to load balance the queries on four Enhydra [9] servers hosting the OpenUSS application. Each Enhydra server relies on a JOnAS J2EE server to access the database tier. All servers are using the C-JDBC driver to access a replicated C-JDBC controller hosting a fully replicated Firebird database.

The operating systems in use at the universities using OpenUSS include Linux, HP-UX and Windows. The database engines include InterBase, Firebird, PostgreSQL and HypersonicSQL. As C-JDBC is written in Java and does not require any application or database changes, it accommodates all of these environments.

### 5.3. Flood alert system

floodalert.org is implementing a replacement for a flood alert system for Rice University and the Texas Medical Center. Geographic distribution in this system is essential, because the system must continue to perform if the two sites are threatened by a flood.

JBoss is used for application level clustering, and C-JDBC provides database clustering of MySQL databases. All nodes are located on a VPN to deal with the security issues resulting from running a cluster over a public network. Horizontal scalability with transparent failover is the most important C-JDBC feature, because the system has to be able to survive the loss of any node at any time. The ability to have database vendor independence was also much appreciated in this project.

Figure 8 gives an overview of the floodalert.org system. There are at least three nodes in the system at all times, each with its own database and application server, and each at a different site. At least one site is several hundred miles from the others for disaster recovery.

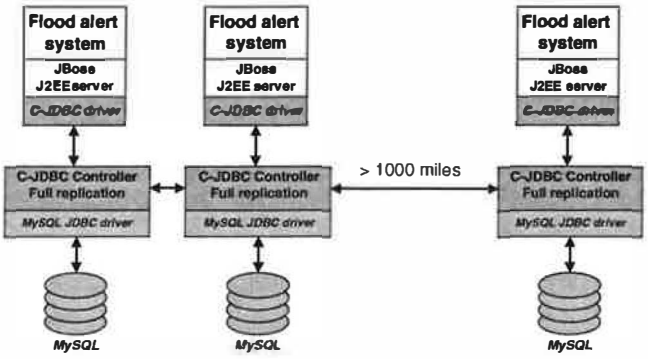



**Figure 8. Flood alert system using C-JDBC**

C-JDBC's ability to dynamically add and remove nodes allows floodalert.org to bring nodes, either new or stale, into the system without much work. Future versions of the system may include bootable CD-ROMs (like Knoppix or Gentoo LiveCD) that will allow floodalert.org to quickly add a node to the system from any computer with an internet connection.

### 5.4. J2EE cluster benchmarking

JMOB (Java Middleware Open Benchmarking) [12] is an ObjectWeb initiative for benchmarking middleware. When running J2EE benchmarks on commodity hardware, the database is frequently the bottleneck resource [8]. Therefore, increased database performance is crucial in order to observe the scalability of the J2EE server. Figure 9 shows a J2EE cluster benchmarking environment example.

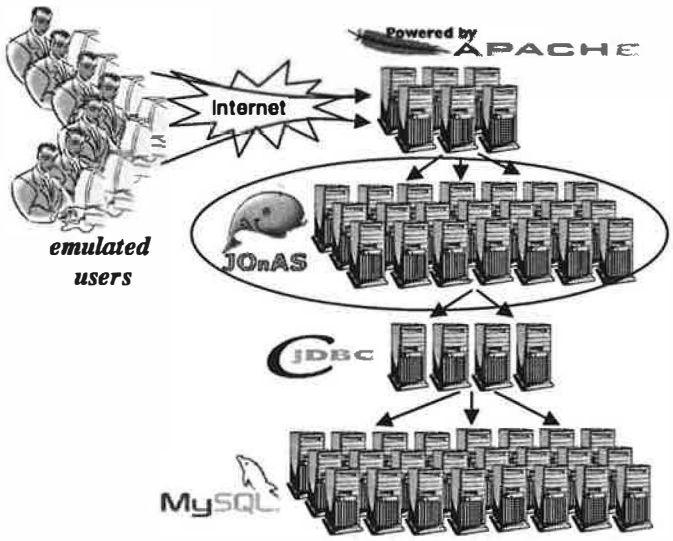



**Figure 9. J2EE cluster benchmarking environment**

A number of emulated users send HTTP requests to a cluster of web servers (Apache in this example). The Apache servers forward the requests to the J2EE cluster under test (JOnAS in this example). The JOnAS cluster accesses a single virtual database that is implemented by a C-JDBC cluster using several controllers and a large number of backends to scale up to the load required by the J2EE cluster.

The vertical scalability and support for partial replication allows large scale configurations providing high performance and sustained throughput. The next section describes a performance evaluation of C-JDBC.

## 6. Performance evaluation

To provide some indication of the performance and the scalability of clusters built using C-JDBC, we describe next a set of experiments carried out with a clustered implementation of the TPC-W benchmark. We also show some results for query response caching using the Rubis benchmark.

### 6.1. Experimental environment

The Web server is Apache v.1.3.22, and Jakarta Tomcat v.4.1.27 [11] is used as the servlet server. We use MySQL v.4.0.12 [14] as our database server with the InnoDB transactional tables and the MM-MySQL v2.0.14 type 4 JDBC driver. The Java Virtual Machine used for all experiments is

IBM JDK 1.3.1 for Linux. All machines run the 2.4.16 Linux kernel.

We use up to six database backends. Each machine has two PII-450 MHz CPUs with 512MB RAM, and a 9GB SCSI disk drive[1]. In our evaluation, we are not interested in the absolute performance values but rather by the relative performance of each configuration. Having slower machines allows us to reach the bottlenecks without requiring a large number of client machines to generate the necessary load. All machines are connected through a switched 100Mbps Ethernet LAN.

### 6.2. The TPC-W benchmark

The TPC-W specification [18] defines a transactional Web benchmark for evaluating e-commerce systems. TPC-W simulates an online bookstore like amazon.com.

Of the 14 interactions specified in the TPC-W benchmark specification, six are read-only and eight have update queries that change the database state. TPC-W specifies three different workload mixes, differing in the ratio of read-only to read-write interactions. The *browsing* mix contains 95% read-only interactions, the *shopping* mix 80%, and the *ordering* mix 50%. The shopping mix is considered the most representative mix for this benchmark.

We use the Java servlets implementation from the University of Wisconsin [4]. The database scaling parameters are 10,000 items and 288,000 customers. This corresponds to a database size of 350MB.

For these experiments C-JDBC is configured without a cache but with parallel transactions and early response to updates and commits. The load balancing policy is Least Pending Requests First.

### 6.3. Browsing mix

Figure 10 shows the throughput in requests per minute as a function of the number of nodes using the browsing mix, for full and partial replication.

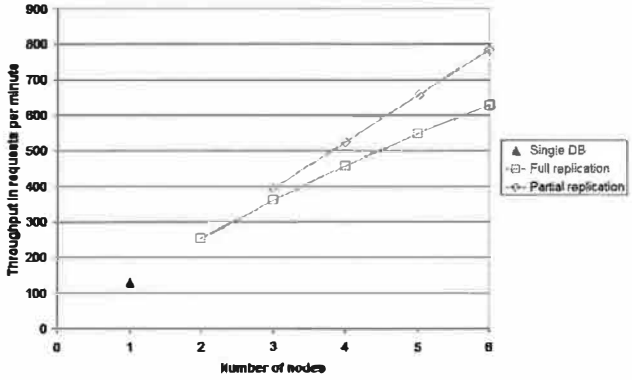



**Figure 10. Maximum throughput in SQL requests per minute as a function of database backends using TPC-W browsing mix.**

[1] These machines are old but they have a CPU vs I/O ratio comparable to recent workstations.

The single database configuration saturates at 129 requests per minute. Full replication starts with a throughput of 251 requests per minute with 2 nodes. The 6-node configuration reaches 628 requests per minute, representing a speedup of 4.9. This sub-linear speedup is due to the MySQL implementation of the best seller query. A temporary table needs to be created and dropped to perform this query. With full replication each backend does so, but only one backend performs the select on this table. Partial replication limits the temporary table creation to 2 backends. Partial replication improves full replication performance by 25% and achieves linear speedup. This example demonstrates the benefit of being able to specify partial replication on a per-table basis.

### 6.4. Shopping mix

Figure 11 reports the throughput in requests per minute as a function of the number of nodes for the shopping mix, which is considered the most representative workload.

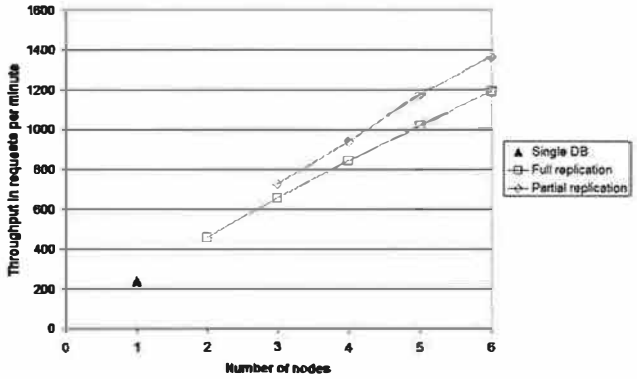



**Figure 11. Maximum throughput in SQL requests per minute as a function of database backends using TPC-W shopping mix.**

The single database without C-JDBC achieves 235 requests per minute at the peak point. Full replication achieves 1,188 requests per minute with 6 nodes. The shopping workload mix scales better than the browsing workload mix due to the smaller number of best seller queries. Partial replication again shows the benefits of per-table partial replication over full replication with a peak throughput of 1,367 requests per minute with 6 nodes.

### 6.5. Ordering mix

Figure 12 shows the results for the ordering mix for partial and full replication. The ordering features 50% read-only and 50% read/write interactions. Even in this scenario, with a large number of group communication messages as a result of the large fraction of updates in the workload, good scalability is achieved.

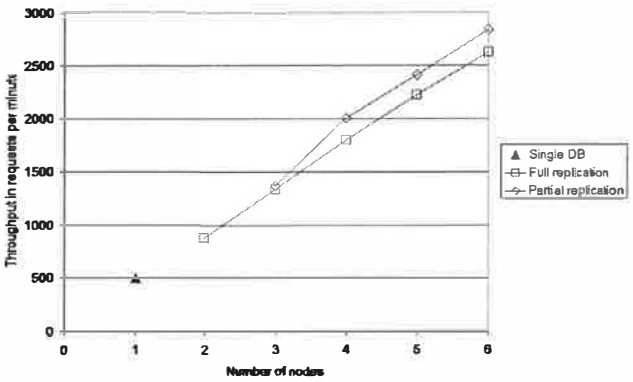



**Figure 12. Maximum throughput in SQL requests per minute as a function of database backends using TPC-W ordering mix.**

Full replication peaks at 2,623 requests per minute with 6 nodes, while partial replication achieves 2,839 requests per minute. The speedups over a single backend are 5.3 and 5.7 for full replication and partial replication, respectively.

### 6.6. Benefits of query result caching

It is also advantageous to use C-JDBC solely for its query result caching feature, even with only a single database backend. We have evaluated the benefits of query result caching using the servlet version of the RUBiS benchmark [1]. The RUBiS (Rice University Bidding System) benchmark models an auction site similar to eBay. We use the bidding mix workload that features 80% read-only interactions and 20% read-write interactions. Table 1 shows the results with and without C-JDBC query result caching.

| *RUBiS bidding mix With 450 clients* | **No cache** | **Coherent cache** | **Relaxed cache** |
|---|---|---|---|
| Throughput (rq/min) | 3892 | 4184 | 4215 |
| Avg response time | 801 ms | 284 ms | 134 ms |
| Database CPU load | 100% | 85% | 20% |
| C-JDBC CPU load | - | 15% | 7% |

**Table 1. RUBiS benchmark (servlet version) performance improvement for a single MySQL backend with C-JDBC query result caching.**

The peak throughput without caching is 3,892 requests per minute for 450 clients. The average response time perceived by the user is 801ms. The CPU load on the database is 100%. With C-JDBC and with consistent query result caching enabled, the peak throughput increases to 4,184 requests per minute, and the average response time is reduced by a factor of almost 3 to 284 ms. The CPU load on the database decreases to 85%,.

Further performance improvements can be obtained by relaxing the cache consistency. With a cache in which content can be out of date for up to 1 minute (entries are kept in the cache for 1 minute independent of any updates), peak throughput reaches 4,215 requests per minute, average response time drops to 134 ms, and CPU load on the database backend is reduced to 20%.

## 7. Conclusion

We presented Clustered JDBC (C-JDBC), a flexible and efficient middleware solution for database replication. By using the standard JDBC interface, C-JDBC works without modification with any application that uses JDBC and with any database engine (commercial or open-source) that provides a JDBC driver.

We have presented several use cases, illustrating how the C-JDBC's flexible configuration framework addresses user concerns such as high availability, heterogeneity support and performance scalability. Combining both horizontal and vertical scalability provide support for large-scale replicated databases. Query response caching improves performance further even in the case of a single database backend.

C-JDBC has been downloaded more than 15,000 times since its first beta release ten months ago. There is a growing community that shares its experience and provides support on the c-jdbc@objectweb.org mailing list. C-JDBC is an open-source project licensed under LGPL and is available for download from http://c-jdbc.objectweb.org.

## 8. Acknowledgements

We would like to thank Peter A. Daly of budget-ha.com for authorizing the reuse of materials from his web site. Lofi Dewanto from OpenUSS was very helpful and supportive of C-JDBC. J. Cameron Cooper provided us with a description of floodalert.org.

We are grateful to the C-JDBC user community for their feedback and support. We would like to thank all C-JDBC contributors who help us improve the software by testing it in various environments and contribute to the code base with fixes and new features.

Finally, we would like to thank our colleagues Anupam Chanda, Stephen Dropsho, Sameh Elnikety and Aravind Menon for their comments on earlier drafts of this paper.

## 9. References

[1] Cristiana Amza, Emmanuel Cecchet, Anupam Chanda, Alan Cox, Sameh Elnikety, Romer Gil, Julie Marguerite, Karthick Rajamani and Willy Zwaenepoel – Specification and Implementation of Dynamic Web Site Benchmarks – *IEEE 5th Annual Workshop on Workload Characterization (WWC-5)*, Austin, TX, USA, November 2002.

[2] Bela Ban – Design and Implementation of a Reliable Group Communication Toolkit for Java – Cornell University, September 1998.

[3] P.A. Bernstein, V. Hadzilacos and N. Goodman – *Concurrency Control and Recovery in Database Systems* – Addison-Wesley, 1987.

[4] Todd Bezenek, Trey Cain, Ross Dickson, Timothy Heil, Milo Martin, Collin McCurdy, Ravi Rajwar, Eric Weglarz, Craig Zilles,

and Mikko Lipasti – Characterizing a Java Implementation of TPC-W – *3rd Workshop On Computer Architecture Evaluation Using Commercial Workloads* (CAECW), January 2000.

[5] Boris Bialek and Rav Ahuja – IBM DB2 Integrated Cluster Environment (ICE) for Linux – IBM Blueprint, May 2003.

[6] Budget-HA.com – High Availability Infrastructure on a budget – http://www.budget-ha.com.

[7] Caucho Technology – Resin 3.0 servlet container - http://www.caucho.com/resin-3.0/

[8] Emmanuel Cecchet, Anupam Chanda, Sameh Elnikety, Julie Marguerite and Willy Zwaenepoel – Performance Comparison of Middleware Architectures for Generating Dynamic Web Content - *Middleware 2003*, ACM/IFIP/USENIX International Middleware Conference, June 2003.

[9] Willy Chiu – Design for Scalability - an Update – *IBM High Volume Web Sites, Software Group*, technical article, april 2001. [9] Enhydra Java application server – http://enhydra.objectweb.org/.

[10] Enhydra Octopus – http://octopus.enhydra.org/.

[11] Jakarta Tomcat Servlet Engine – http://jakarta.apache.org/tomcat/.

[12] JMOB – Java Middleware Open Benchmarking – http://jmob.objectweb.org/.

[13] Bettina Kemme and Gustavo Alonso – Don't be lazy, be consistent: Postgres-R, a new way to implement Database Replication – *Proceedings of the 26th International Conference on Very Large Databases*, September 2000.

[14] MySQL Reference Manual – MySQL AB, 2003.

[15] OpenUSS – Open Source Software for Universities and Faculties (Open Source University Support System) - http://openuss.sourceforge.net/.

[16] Oracle – Oracle9*i* Real Application Clusters – Oracle white paper, February 2002.

[17] D. Stacey – Replication: DB2, Oracle or Sybase – *Database Programming & Design 7*, 12.

[18] Transaction Processing Performance Council – http://www.tpc.org/.

[19] S. White, M. Fisher, R. Cattel, G. Hamilton and M. Hapner – *JDBC API Tutorial and Reference, Second Edition* – Addison-Wesley, ISBN 0-201-43328-1, november 2001.

# Wayback: A User-level Versioning File System for Linux

Brian Cornell Peter A. Dinda Fabián E. Bustamante
*Computer Science Department, Northwestern University*
{techie,pdinda,fabianb}@northwestern.edu

## Abstract

In a typical file system, only the current version of a file (or directory) is available. In Wayback, a user can also access any previous version, all the way back to the file's creation time. Versioning is done automatically at the write level: each write to the file creates a new version. Wayback implements versioning using an undo log structure, exploiting the massive space available on modern disks to provide its very useful functionality. Wayback is a user-level file system built on the FUSE framework that relies on an underlying file system for access to the disk. In addition to simplifying Wayback, this also allows it to extend any existing file system with versioning: after being mounted, the file system can be mounted a second time with versioning. We describe the implementation of Wayback, and evaluate its performance using several benchmarks.

## 1 Introduction

A user of a modern operating system such as Linux experiences a very simple file system model. In particular, the file system only provides access to the current versions of his or her files and directories. The trouble with this model is that the user's progress in his work is not monotonic – the user makes mistakes such as discarding important text in a document, damaging carefully tuned source code through misunderstanding, or even accidentally deleting files and directories. Beyond mistakes, it is often helpful, especially in writing code or papers to look at the history of changes to a file. If the user could "go back in time" (using the "wayback machine" from Ted Keys's classic cartoon series "Peabody's Improbable History," for example), she would be in a better position to recover from such mistakes or understand how a file got to its current state.

Of course, version control systems such as RCS [RCS] and CVS [CVS] provide such functionality. However, the user must first become familiar with these systems and explicitly manage her files with them. In particular, the user must tell these systems when a new version of the files is ready by explicitly committing them. Hence, the user determines the granularity of versions, and, since he must explicitly make them, they tend to be large. Some tools such as EMACS include drivers to automate this process. Some applications (e.g. Microsoft Word) provide their own internal versioned file format. Here the versioning is usually done automatically at the granularity of a whole editing session.

We believe that a better way to help the user revert to earlier versions of her file is to automatically provide versioning in the file system itself, and to provide it at a fine granularity. To this end, we have developed the Wayback versioning file system for Linux. With no user interaction, Wayback records each write made to each file or directory into a permanent undo log [UNDO]. The undo log can then be unwound in reverse order (prompted by a user-level tool) to rollback to or extract any previous version of a file. Wayback is implemented as a user-level file system using the FUSE kernel module [FUSE]. It operates on top of an existing, non-versioned file system, adding versioning functionality.

Versioning file systems have been around for quite some time. It was already provided by some early file systems such as Cedar File System [CEDAR] and 3-DFS [3DFS]. The VMS operating system from DEC introduced versioning to a broad range of users. The VMS file system created a new version of a file on each close [VMS]. Checkpointing is an alternative approach to provide versioning: snapshots of the entire file system are taken periodically and made available to the user. Example of systems using checkpointing include AFS [AFS], Petal [Petal] and Ext3Cow [Ext3Cow]. One limitation of checkpoint-based versioning is that changes made between checkpoints cannot be undone. The Elephant versioning file system was among the first to recognize that versioning was a excellent way to exploit the massive (and exponentially growing) disk sizes that are available today [Elephant]. There has since been an explosion of interest in versioning file systems. Wayback does versioning at the level of writes and hence is a *comprehensive* versioning file

system [CVFS]. Its ability to run on top of an existing file system is similar to the concept of a stackable file system such as Versionfs [VersionFS2]. Versionfs implements some of the same functionality of Wayback on Linux [VersionFS1], but there is no public release available. As far as we are aware, Wayback is the first public release (under the GPL) of a versioning file system for Linux.

## 2 User's View Of The Wayback FS

Wayback FS requires a recent 2.4 or 2.6 Linux kernel, gcc 2.95.2 or higher, and Perl 5. We have used kernel versions as early as 2.4.7-10 (shipped with Red Hat 7.2). The FUSE user-level file system module is used (versions 0.95 and up). The current Wayback distribution is shipped along with the FUSE source. Compilation involves the typical make, make install routine. The output includes:

- fuse.o: FUSE kernel module
- wayback: Wayback FS user-space server
- vutils.pl: command-line utility for manipulating files.
- mount.wayback: easy mounting script

Four symbolic links to vutils.pl are also created to expose its basic functions:

- vstat: Describe a versioned file.
- vrevert: Revert a versioned file to an earlier version.
- vextract: Extract a specific version of a file.
- vrm: Permanently remove a file.

To mount a Wayback file system, the underlying file system is first mounted in the normal manner, then it is remounted by starting a Wayback server:

$ wayback path-in-underlying-fs mount-path

A script named mount.wayback is included in the distribution that remounts paths nicely such that all users can access the versioned files as they could the underlying files. mount.wayback is executed with the same options as wayback above.

After this, the user can access his files through mount-path. Any change made will be logged and is reversible using vrevert. Old versions can also be copied out using vextract. Even "rm" is logged and can be undone. To permanently remove a file, vrm is used. Notice that it is possible to mount the directory hierarchy under any path as a new, versioned, file system. It is also possible to continue to manipulate files in the original path, but those changes will not be logged and are not revertible.

Versions are tagged in two ways: by change number (starting with one being the most recent change) and by time stamp. The user most often uses the time stamp, it being natural to revert or extract a file or directory as it existed at a particular point in time.

## 3 Implementation

Wayback FS is implemented as a user-space server that is called by the FUSE kernel module. In essence, FUSE traps system calls and upcalls to the Wayback server. The server writes an entry into the undo log for the file or directory that reflects how to revert the effects of the call, and then executes the call on the underlying file system. We opted for FUSE because of familiarity with the tool. We could have alternatively employed SFS [FiST].

### 3.1 Log Structure For Files

Each file for which versioning information exists has a shadow undo log file, named by default "<filename>. versionfs! version". Each log record consists of:

- A time stamp,
- The *offset* at which data is being overwritten or truncated,
- The *size* of the data that is being lost,
- Whether the file size is increasing due to writing off the end of the file, and
- The *data* being lost, if any.

### 3.2 Logging File Operations

Wayback traps calls to write() and truncate() for files. Every time write() is called on a file, versionfs reads the portion of the file that the write call will overwrite and records it in the log. The *offset* recorded is the offset in the file at which data is being written, the *size* is either the number of bytes to the end of the file or the number of bytes being written (whichever is smaller), and the data is the *size* bytes at *offset* that are currently in the file.

When truncate() is called on a file, the *offset* recorded is the length to which the file is truncated, *size* is the number of bytes in the file after that point, and *data* is that data that is being discarded due to truncation.

### 3.3 Log Structure For Directories

Every directory has a shadow undo log that we call the directory catalog. The directory catalog logs when any entry in the directory is created, removed, renamed, or has its attributes change. The directory catalog has the default name "<directory>/. versionfs! version". Each log record consists of:

- A time stamp,
- The *operation* being performed,
- The *size* of the data recorded for this operation, and
- The *data* needed to undo the operation. The interpretation of the data depends on the operation.

### 3.4 Logging Directory Operations

When mknod(), mkdir(), or creat() is called, a link is created, or open() is called with the O_CREAT flag, the directory catalog is updated with the create or mkdir operation number and *data* consisting of the filename that is being created.

When unlink() is called on a regular file, the file is first truncated to zero length to preserve the contents of the file before deletion. Next, the directory catalog is updated with *data* consisting of the attributes of the file (mode, owner, and times) and the filename that is being deleted. For links, the destination is also recorded.

Calls to rmdir() in Wayback actually translate to calls to rename(). Directories are *never* deleted because their contents would be lost. Instead an identifier such as ". versionfs! deleted" is added to the directory name. Subsequently, and for user-initiated rename() calls, the directory catalog is updated with *data* consisting of the old name of the file or directory, and the new name of the file or directory.

When chmod(), chown(), or utime() is called, the directory catalog is updated with *data* consisting of the attributes of the file and the filename for which attributes are being changed.

## 4 Design Issues

We encountered several issues while designing and implementing Wayback. The solutions we found and decisions we made have defined what Wayback is now.

### 4.1 Kernel Versus User-level

The first major decision we had to make was whether this file system should be implemented in the kernel as its own file system, or using a user-level module. The trade-offs are in speed, ease of implementation, and features. A kernel module would undoubtedly be much faster because the user-level overhead would be avoided. However, it could limit compatibility to certain kernel versions, and it would preclude adding versioning to existing file systems. It would also be much harder to implement a kernel module.

The main factor in our decision to make a user-level file system had to do with the features we could easily implement. We considered writing Wayback as a kernel-level extension to ext3. This would probably have been faster, but it would have been limited to ext3 file systems on normal block devices. Implementing Wayback as a user-level file system would make it slower, but would let us remount any file system with versioning.

### 4.2 Choice of Undo Logging

Wayback logs changes as undo records. We recover previous versions by applying these records in reverse order until the appropriate version is reached. This is straightforward, but it has a downside: while reverting to newer versions is very fast, reverting to very old versions can take some time. One alternative is a redo log, in which modifications themselves are written as log records. Recovering an old version means applying the records in forward order until the appropriate version is reached. This has the advantage of allowing very old versions to be recovered very quickly, but newer versions are slow. A third possibility is an undo/redo log, which contains both undo and redo records, allowing us to move backward and forward easily. Each logging technique can be combined with periodic checkpointing, providing snapshots of the whole file state from which to move forward or backward using the log.

We chose simple undo logging for Wayback because we felt that for our use cases – reverting mistakes made in editing programs and documents – we would typically have only to move backward by a small number of versions. In light of other applications we would like to support (see Conclusion), we are reconsidering our logging model.

### 4.3 Use of FUSE

Once we had decided on a user-level approach, we next considered how to interact with the kernel. At the time we started development, FUSE was still in its early stages (we started with FUSE 0.95), but being able to avoid kernel development altogether was very tempting to us since we wanted to concentrate on the versioning mechanisms. The FUSE kernel module provided us with the level of access we needed on a modern Linux kernel. FUSE proved to be relatively stable and easy to use.

Early versions of FUSE did not provide an upcall for the close() system call. This lack would have made it impossible to create new versions on close, as in VMS. Fortunately, we had determined to do write-level versioning. However, it still indirectly affected Wayback's design. In particular, without close() calls, managing file descriptors for log files is made unnecessarily difficult.

### 4.4 Path Redirection

After deciding to use FUSE, we quickly came upon another issue. FUSE is designed only to provide a destination path for the file system, and not a source path to mount there. The examples for FUSE either remount an entire directory structure beginning in the root directory, or provide their own root from a different source such as memory.

We decided that we wanted to have redirection from any path, not just the root directory, so we had to implement a work-around. Wayback takes different command-line arguments from other FUSE file systems, and then modifies those arguments before passing them on to FUSE. We then use the information from those arguments to modify every path given to Wayback from FUSE, redirecting it to the "real" path.

## 5 Performance Evaluation

A variety of performance tests were run on Wayback FS to evaluate its performance. These tests include Bonnie [Bonnie], which performs various operations on a very large file; the Andrew Benchmark [Andrew], which simulates the operation of normal file systems; and a test that compares using Wayback FS to using manual checkins with RCS.

These tests were all run on three test systems:

- Machine A: AMD Athalon XP 2400+ with 512 MB of RAM using an internal 2.5 inch notebook hard drive.
- Machine B: Intel Pentium 4 2.2 GHz with 512 MB of RAM using an external USB 1.1 disk (1.5 MBps).
- Machine C: Intel Celeron 500 MHz with 128 MB of RAM using an internal 2.5 inch notebook hard drive.

All of the tests were run under Linux kernel 2.4.20 or 2.4.22.

For comparison, another file system was built on top of FUSE before tests were run. This file system simply redirects requests from the mounted file system through to the underlying file system, acting as a pass-through. This file system is used to identify the performance hit taken solely by the FUSE system, and isolate the performance loss from versioning. We consider the following file systems:

- ext3: the out-of-box ext3 file system
- ext3+fuse: ext3 run through our pass-through FUSE file system
- ext3+ver: Wayback FS running on top of ext3

These configurations were used for our Bonnie and Andrew tests. In comparing to RCS, we compared "ext3+ver" to "ext3", where the files in question were periodically committed using RCS.

We did not run any tests on the performance of reverting files, because it is not an everyday occurrence and shouldn't matter as much as reading and writing performance. Reverting or extracting a recent state from a file typically takes at the most seconds if not less than a second. Disk space usage does increase when reverting files depending on the size of the reversion, because Wayback does not remove the reverted entries from the log. Rather it runs them backwards on the file, creating more entries in the log.

### 5.1 Bonnie

Bonnie was originally created by Tim Bray to identify bottlenecks in file systems. It does so by creating a very

large file, large enough to require disk access rather than caching, and reading from/writing to that file as fast as it can.

### 5.1.1 Bonnie Implementation

Bonnie is implemented as a single C program. The program takes arguments for the size of the file it will use and the path where it will be stored. Bonnie then creates the large file and performs many sequential operations on it, including writing character by character, writing block by block, rewriting, reading character by character, reading block by block, and seeking. For this test, Bonnie was run with a file size of 1 GB.

### 5.1.2 Bonnie Results

Figures 1-3 show the performance of the different Bonnie phases on the three machines. For each phase and machine, we compare ext3, ext3 via our pass-through FUSE file system, and ext3 with Wayback versioning. The performance metric is in KB/s as measured by Bonnie. The point is to measure how much performance is lost by using Wayback and how it breaks down into FUSE overheads and the actual costs of Wayback. Figure 4 shows the CPU costs, in terms of percentage of CPU used as measured by Bonnie.

It is important to point out that in some cases layering ext3 below FUSE actually increases its performance. We expect that this is due to buffering effects as there is now an additional process which can buffer. Additionally, the overheads shown in Figure 4 are slightly misleading. Bonnie is measuring the system and user

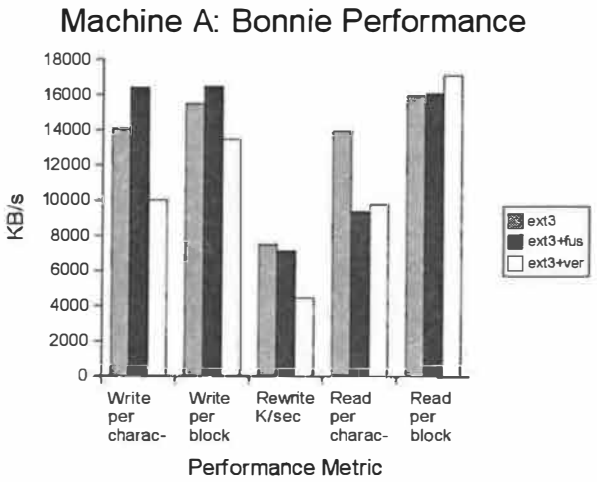



*Figure 1. Bonnie Performance on Machine A*

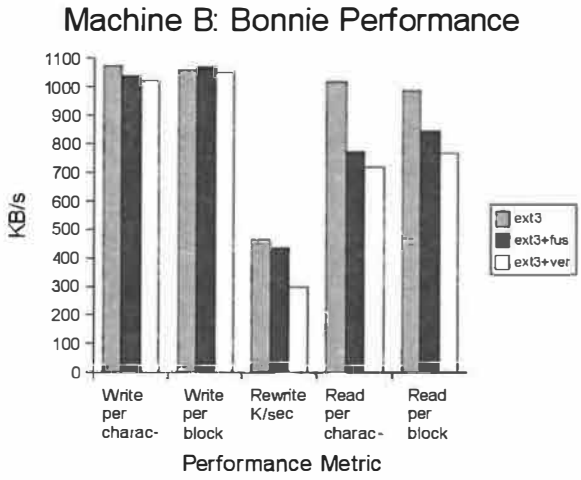



*Figure 2. Bonnie Performance on Machine B*

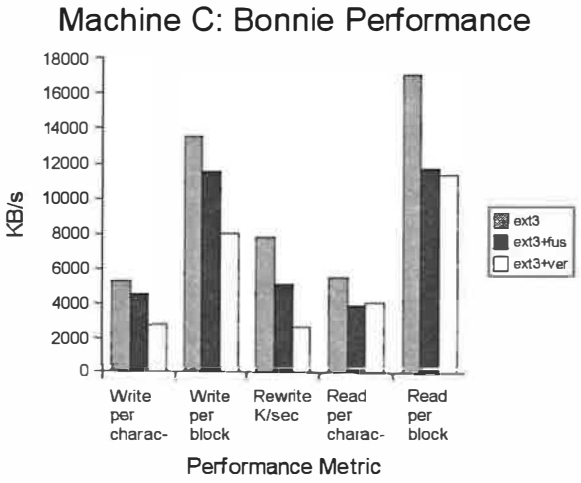



*Figure 3. Bonnie Performance on Machine C*

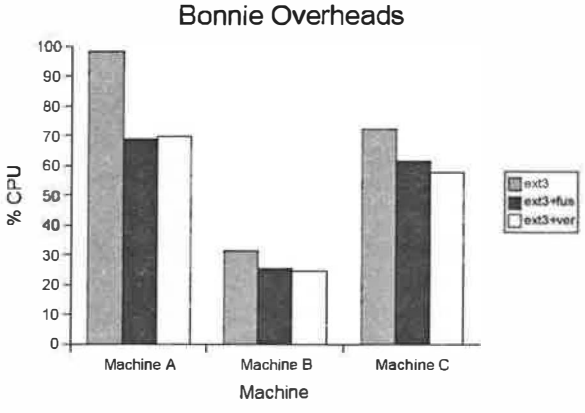



*Figure 4. Bonnie Overheads*

time it uses, but does not count the time spent in the Wayback server on its behalf.

For block writes in Wayback, we see performance impacts in the range of -2% to -40% compared to un-

adorned ext3, depending on the speed of the disk and the machine. For block reads, the performance impact is +5% to -32%. Character writes are impacted -3% to -50%, while the character read impact is -10% to -30%. In re-writing, where would expect to see the maximum impact, the range is -30% to -70%.

The largest impact on write speed is on machines with fast disks, particularly those that also have slow processors. The largest impact on rewrite speed is on machines with slow disks, which is to be expected as rewrites will include additional data to be written to the logs. Read speed is maximally affected on slow machines with fast disks. In many cases, a large portion, often the majority of the performance impact is due to FUSE rather than versioning.

## 5.2 Andrew Benchmark

The Andrew Benchmark, although quite old, is commonly used to measure the performance of file systems. It performs operations similar to those performed every day on file systems, including making directories, copying files, stating and reading files, and compiling a program.

### 5.2.1 Andrew Implementation

The original Andrew Benchmark was written on and designed for Sun computers. It consists of a makefile that executes operations in five phases, ending in the compilation of a program. The program used in the benchmark will only compile on Sun systems however. The Andrew benchmark also only runs each phase once, and does not delete the files it creates.

Because of these limitations, we rewrote the Andrew Benchmark in Perl. The program runs the same phases as the original Andrew Benchmark, except that it can run them with any set of files. It can also run the test multiple times and print a summary.

The phases of the Andrew Benchmark are designed to emulate everyday use of a file system. The phases are all done using the source directory of a program, and include:

- Phase 1: Create five copies of the directory structure in the source tree.
- Phase 2: Copy all of the files from the source tree into the first set of directories created.
- Phase 3: Stats each file using `ls -l`.
- Phase 4: Read each file using grep and wc.
- Phase 5: Compile the source in the test tree.

The source that we used is that of the window manager ION. Each phase was executed 1000 times to get accurate results. As before, we ran the benchmark three times on each time, once with the ext3 file system, once with the pass-through file system on ext3, and once with Wayback FS on ext3.

### 5.2.2 Andrew Results

Figures 5-7 compare the performance of the different file systems for each phase on each machine. The performance metric is the average wall-clock time to run each phase. Phase 5 (Compilation) times have been divided by 20 to fit on the graphs.

There are several takeaway points from these graphs. First, the largest performance impact of Wayback is on directory creation (Phase 1). Second, Wayback increases the time to run the write-intensive copy phase (Phase 2) by between a factor of two and a factor of four. The largest impact is, not surprisingly, on a machine with a slow disk. Wayback has negligible impact on the stat phase (Phase 3), except on very slow machines. The impact on reads (Phase 4) is relatively low (30%) regardless of the machine or disk. Finally,

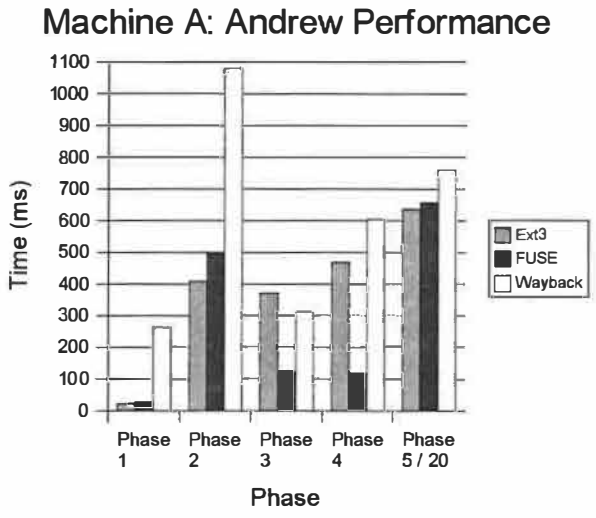



*Figure 5. Andrew Performance on Machine A*

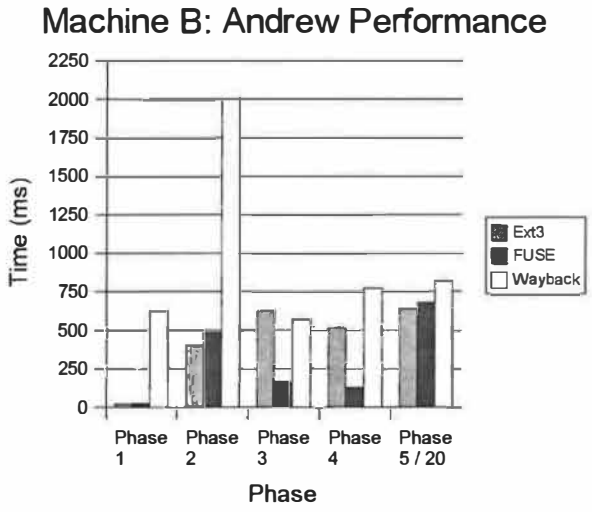



*Figure 6. Andrew Performance on Machine B*

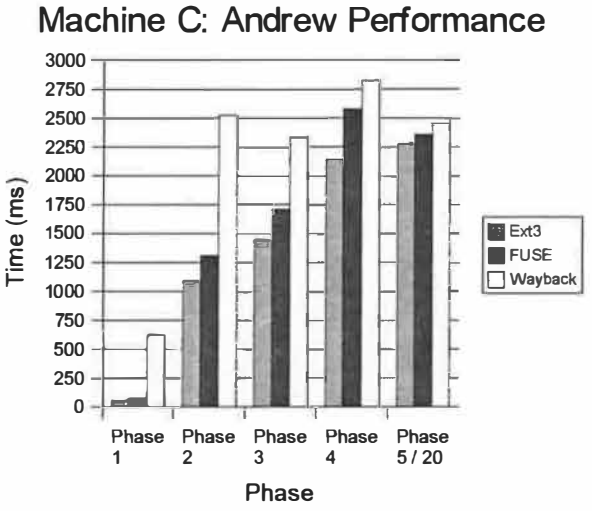



*Figure 7. Andrew Performance on Machine C*

for compilation (Phase 5), the impact of Wayback is very small (15%) on all three machines.

That the performance impact on compilation is marginal suggests that Wayback could be used very effectively in the edit-compile-debug loop of program development or document preparation with tools such as LaTeX.

## 5.3 RCS Comparison

In order to test the effectiveness of a versioning file system, it is necessary not only to compare it to other file systems, but to compare it to other methods of versioning. For this reason we have constructed a test that compares different operations on Wayback FS with similar operations using RCS on an ext3 file system.

### 5.3.1 RCS Implementation

The RCS comparison is implemented as a Perl script that runs through a variety of tests multiple times on both an RCS system and Wayback. The test records the time taken in each case as well as the disk space used.

The RCS comparison runs three modes of testing and produces separate output for the three modes:

- Mode 1: Random seeks within a binary file followed by writing a specified amount of random data. This is designed to emulate normal binary file use. We used 1 MB binary files and 1 KB writes.

- Mode 2: Read an entire binary file into memory, change a specific number of randomly chosen locations with a specified amount of random data, then write the file back to disk. This is similar to the operation of some databases. For this test we used 1 MB files, 1 KB writes, and randomly between 5 and 20 writes per iteration.

- Mode 3: Randomly choose a line in a text file, change a specified number of lines randomly using English words, truncate the file and write everything after the point at which it began changing lines. This test uses a dictionary file to construct files. This is designed to emulate text editing, including changing configuration files and writing code. For this test we used files of 2000 lines, 20 words maximum per line, and changed randomly between 1 and 5 lines per iteration.

Each mode in this test was run for 100 iterations with a file, and the whole ensemble was repeated 10 times with different files. Wayback logged every operation as normal. For RCS, we committed the file periodically, varying the period.

### 5.3.2 RCS Results

Figures 8-10 show our results. As before, each Figure corresponds to a particular machine. Three curves, one for each mode, are included. Times for the third mode have been multiplied by 10 to fit on the graphs. The vertical axis is the time required to run the mode, while the horizontal axis is the test set. The left-most test set, marked "Version" is for Wayback. The remaining test sets are for RCS with varying period. For example, "RCS 1" corresponds to RCS commits done after every operation, which is equivalent to what Wayback is doing, while "RCS 6" corresponds to RCS commits done

after every 6th operation. As the period increases, we amortize more and more of the overhead of using RCS and get closer to the performance of Wayback.

It is clearly the case that Wayback performs far better than RCS at comprehensive versioning. An interesting trend is that for slower processors the difference between Wayback and RCS is greater, and for slower disks RCS nearly catches up to Wayback.

The results show that except in the case of a very slow disk, Wayback performs better with single binary writes (Mode 1) even if RCS is used with a period of 10. On an average system, Wayback performs about as well as using RCS every other time when writing the whole binary file (Mode 2). Using Wayback with text on an average system is similar in performance to using RCS about once every four changes.

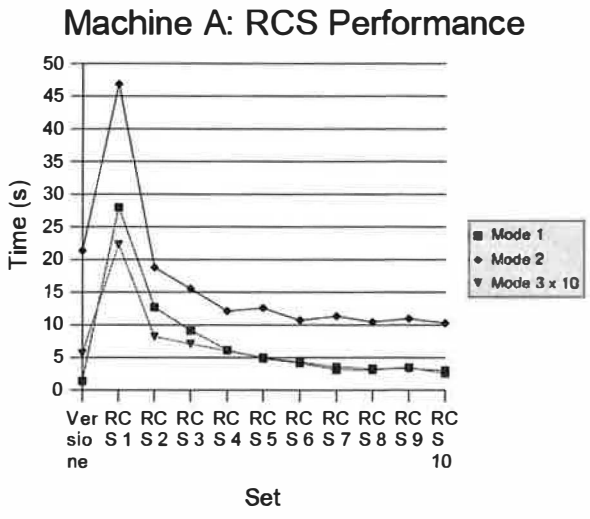



*Figure 8. RCS Performance on Machine A*

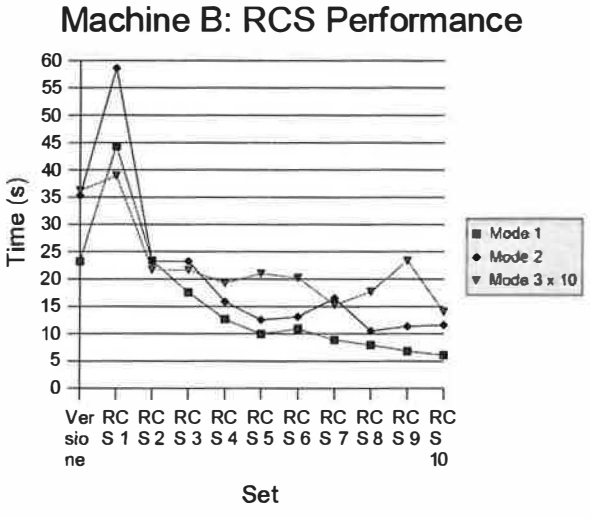



*Figure 9. RCS Performance on Machine B*

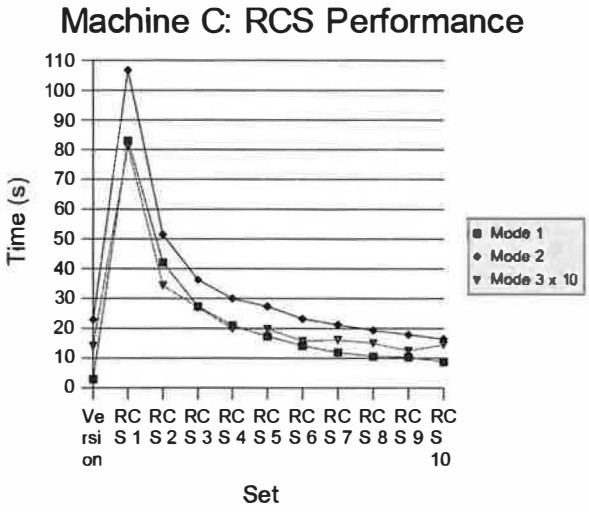



*Figure 10. RCS Performance on Machine C*

In terms of disk space use, the results are quite different. For the single binary writes (Mode 1), Wayback uses much less disk space than RCS. For writing the whole binary file (Mode 2), Wayback uses 25 to 30 times as much space as RCS. For text changes (Mode 3) Wayback uses about 20 times as much space as RCS. These results are summarized in Table 1; sizes are shown in bytes and are the average from 10 runs. Disk space is not dependent on the test system, so results are only shown from Machine A.

| File Type | Mode 1 | Mode 2 | Mode 3 |
|---|---|---|---|
| Versioned | 1157456.4 | 106347428.0 | 2182218.0 |
| RCS Period 1 | 2242325.4 | 3856521.8 | 101062.2 |
| RCS Period 2 | 2237020.7 | 3779180.4 | 96134.2 |
| RCS Period 3 | 2233854.1 | 3731427.8 | 94336.4 |
| RCS Period 4 | 2234384.4 | 3719578.2 | 93597.1 |
| RCS Period 5 | 2233716.4 | 3700853.4 | 93095.3 |
| RCS Period 6 | 2227924.3 | 3621657.1 | 92375.5 |
| RCS Period 7 | 2230300.3 | 3635107.2 | 92321.2 |
| RCS Period 8 | 2227552.2 | 3590195.5 | 91960.0 |
| RCS Period 9 | 2231124.4 | 3625548.8 | 92060.8 |
| RCS Period 10 | 2232218.7 | 3629717.4 | 92045.2 |

*Table 1. RCS Storage Costs*

## 6 Conclusions

We have described the design and implementation of Wayback, a comprehensive versioning file system for Linux. Wayback is implemented as a user-level file system using FUSE. When running on top of the standard Linux ext3 file system, its overhead is quite low for common modes of use.

We are considering several extensions and applications for Wayback. First, if the underlying file system does not support transactional writes, they could be forced by Wayback through sync operations. Second, it appears that a file system that never garbage collects its undo log would naturally perform very well when running on top of, or incorporated into a log-structured file system [Log]. Third, if Wayback used an undo/redo log, it would be straightforward to go forward in time as well as backward. Fourth, hierarchical version numbers and undo/redo logging would permit branching. Of course, it is not clear whether it would be any less painful to handle merging in the file system than in, say, CVS. Finally, given undo/redo logs and version numbers, keeping large files synchronized among multiple sites would be simplified – we would have only to transfer the redo log records that the remote log did not already have and then redo them. For situations where a single large file migrates among multiple sites but is accessed at one site at a time – virtual machine image migration for example – such synchronization might prove to provide dramatically faster migration times.

## Availability

Wayback is publically available (under the GPL) from http://sourceforge.net/projects/wayback.

## References


[3DFS] D.G. Korn and E. Krell. The 3-D File System. In *Proc. of the USENIX Summer Conference,* pp. 147-156, 1989.

[AFS] J. J. Kistler and M. Satyanarayanan. Disconnected operations in the Coda file system. In *Proc. of the 13th ACM Symposium on Operating Systems Principles.* October, 1991.

[Andrew] J. Howard, et al, *Scale and Performance in a Distributed File System,* Transactions on Computer Systems, Volume 6, February 1988.

[Bonnie] T. Bray, *The Bonnie Benchmark*, http://www.-textuality.com/bonnie

[CEDAR] D. K. Gifford, R.M. Needham, and M.D. Schroeder. The Cedar File System. *Communication of the ACM*, 31(3):288-298, 1988.

[CVS] B. Berliner and J. Polk. Concurrent Versions Systems (CVS). http://www.cvshome.org .2001.

[CVFS] C. A. Soules, G.R. Goodson, J. D. Strunk and G. R. Ganger. Metadata Efficiency in a Comprehensive Versioning File System. In *Proc. of the 2nd USENIX Conference on File and Storage Technologies,* 2003

[Elephant] D. S. Santry, M.J. Feeley, N.C. Hutchinson, A.C. Veitch, R.W. Carton and J. Ofir. Deciding When to Forget in the Elephant File System. In *Proc. of the 17th ACM Symposium on Operating System Principles.* December, 1999.

[Ext3Cow] Z. N. Peterson and R. C. Burns. Ext3cow: The design, Implementation, and Analysis of Metadata for a Time-Shifting File System. Tech. Report HSSL-2003-03, Computer Science Department, The John Hopkins University, 2003.

[FiST] E. Zadok and J. Nieh. FiST: A Language for Stackable File Systems. In *Proc. of the Annual USENIX Technical Conference.* June 2000.

[FUSE] M. Szeredi. Filesystem in USEr space,. http://sourceforge.net/projects/avf, 2003.

[Log] M. Rosenblum and J. Ousterhout, The Design and Implementation of a Log-Structured File System, *ACM Transactions on Computer Systems,* 10(1), 1992, 26-52.

[Petal] E. K. Lee and C. A. Thekkath. Petal: Distributed virtual disks. In *Proc. of the 7th Conference on Architectural Support for Programming Languages and Operating Systems.* 1996.

[RCS] F. Tichy, *Software Development Control Based On System Structure Description*, PhD. Thesis, Carnegie Mellon University, 1980.

[UNDO] H. Garcia-Molina, et al, *Database Systems: The Complete Book*, Chapter 17, Prentice Hall, 2002.

[VersionFS1] Muniswamy-Reddy, et al, A Versatile and User-Oriented Versioning File System, In Proc. Of the 3rd USENIX Conference on File Storage and Technologies, March, 2004.

[VersionFS2] *A Stackable Versioning File System*, http://www.fsl.cs.sunysb.edu/project-versionfs.html

[VMS] K. McCoy, *VMS File System Internals*, Digital Press, 1990.

# Glitz: Hardware Accelerated Image Compositing using OpenGL

Peter Nilsson
*Department of Computing Science*
*Umeå University, Sweden*
c99pnn@cs.umu.se

David Reveman
*Department of Computing Science*
*Umeå University, Sweden*
c99drn@cs.umu.se

## Abstract

In recent years 2D graphics applications and window systems tend to use more demanding graphics features such as alpha blending, image transformations and anti-aliasing. These features contribute to the user interfaces by making it possible to add more visual effects as well as new usable functionalities. All together it makes the graphical interface a more hospitable, as well as efficient, environment for the user.

Even with today's powerful computers these tasks constitute a heavy burden on the CPU. This is why many proprietary window systems have developed powerful 2D graphics engines to carry out these tasks by utilizing the acceleration capabilities in modern graphics hardware.

We present Glitz, an open source implementation of such a graphics engine, a portable 2D graphics library that can be used to render hardware accelerated graphics.

Glitz is layered on top of OpenGL and is designed to act as an additional backend for cairo, providing it with hardware accelerated output.

Furthermore, an effort has been made to investigate if the level of hardware acceleration provided by the X Window System can be improved by using Glitz to carry out its fundamental drawing operations.

## 1 Introduction

There is a trend visible in the appearance of modern window systems and 2D graphics in general these days. They all become more and more loaded with graphical features and visual effects for each available product generation.

Unfortunately, these features means heavy computations that takes a lot of time when carried out by the general CPU. In the past this has meant a slowdown throughout the entire system as well as a significant limitation in the kind of visual effects that could be used.

In the field of 3D graphics, similar problems have been solved by implementing the drawing operations in dedicated 3D-graphics hardware. Hardware accelerated rendering means that the graphics hardware contains its own processor to boost performance levels. These processors are specialized for computing graphical transformations, so they achieve better results than the general purpose CPU. In addition, they free up the computer's CPU to execute other tasks while the graphics accelerator is handling graphics computations.

Modern window systems have developed 2D-graphics engines, which utilize the high performance rendering capabilities inherent in today's 3D-graphics hardware. In fact, much of the visual effects and advanced graphics that can be seen in these systems would not even be feasible without hardware accelerated rendering.

This paper presents Glitz, an open source implementation of a 2D graphics library that uses OpenGL[17] to realize a hardware accelerated high performance rendering environment.

Furthermore, these ideas have been applied to the X Window System (X)[16], to see if they could improve hardware acceleration of graphical applications and thereby making way for more advanced graphics.

The software that is developed in this project will primarily target users equipped with the latest in modern graphics hardware.

### 1.1 Traditional X Graphics

X was not originally designed for the advanced graphics that can be seen on modern desktops. The main reason is that graphics hardware available at the time was not fast enough. Application developers soon found the core graphics routines inadequate and the trend became to use client-side software rendering instead. Several steps have been taken to rectify this since then.

One major improvement was made with the introduction of the X Render Extension (Render)[11]. Render has widely been accepted as the new rendering model for X. It brought the desired graphics operations to the applications and thereby filled in the gaps of the core protocol. Some of the features that Render supports are alpha compositing, anti-aliasing, sub-pixel positioning, polygon rendering, text rendering and image transformations. The core of Render is its image compositing model, which borrows fundamental notions from the Plan 9 window system[12]. Render provides a unified rendering operation, which supports the Porter-Duff[13] style compositing operators. All pixel manipulations are carried out through this operation. This provides for a simple and consistent model throughout the rendering

system.

Render allows us to perform advanced graphics operations on server-side. Graphics operations that are performed on server-side can be accelerated by graphics hardware. XFree86's[9] Render implementation uses XFree86 Acceleration Architecture (XAA)[19] to achieve hardware accelerated rendering. XAA breaks down complex Render operations into simpler ones and accelerates them if support is provided by the driver, otherwise it falls back on software. To fall back on software means that all graphics computations are processed by the CPU. For most XFree86 drivers, image data lives in video memory, so for the CPU to be able to access this data it must first be fetched from video memory. The CPU can then perform its computations and the image data must then be transfered back to video memory. The result of such a software fall-back is most likely to be slower than if the operation would have been done on client-side with all image data already local to the CPU.

The ideal situation would be to have XAA Render hooks for all Render operations in every driver. This requires graphics driver developers to add XAA Render hooks in each driver, which results in a duplicated effort. Unfortunately, not many drivers have much support for XAA Render hooks at this point. This results in inconsistent acceleration between different rendering operations, different drivers and different hardware.

## 1.2 Glitz Fundamentals

The Render model seems to be ideal to build Glitz upon. It provides the necessary operations needed to carry out the rendering for modern 2D graphics applications. Hence Glitz is designed to exactly match the Render model semantics, adding an efficient and more consistent hardware acceleration of the rendering process.

The Render model provides only low level fundamental graphics operations, not always suitable for direct use by application developers. A higher level graphics API is needed on top of the Render model to make it useful for this purpose. The cairo library (formerly known as Xr[20]) is a modern, open source, cross-platform 2D graphics API designed for multiple output devices. With its PDF[3]-like 2D graphics API, it provides an attractive and powerful vector based drawing environment. Cairo uses a backend system to realize its multiple output formats. One thing missing thus far in cairo, is a backend that efficiently accelerates the rendering process with today's high performance graphics hardware. The backend interface of cairo has the same semantics as Render. Thus Glitz is designed to act as an additional backend for cairo providing this hardware accelerated output.

The output of Glitz is accelerated in hardware by using OpenGL for all rendering operations. Figure 1 illustrates these ideas by showing the layers involved when an application uses cairo to draw hardware accelerated graphics through Glitz.


*Figure 1: Different layers involved when an application uses cairo to render graphics to an OpenGL surface.*

OpenGL can be used to accelerate 2D graphics output, just as with 3D. Most people think of OpenGL as a 3D graphics API, which is understandable because it was used primarily for 3D applications like visualizations and games in the past. However, it is just as well suited for 2D graphics of the nature discussed in this paper, where transformations and other visual effects play a central part, much like in traditional 3D applications. OpenGL is the most widely used and supported graphics API available today, it is well supported in hardware and very portable. It operates on image data as well as geometric primitives and offers the necessary operations needed for the creation of Glitz.

To sum up these ideas, Glitz is created to act as an interface on top of OpenGL, which provides the same consistent rendering model as Render. This interface is implemented in such a way that it takes advantage of the OpenGL hardware acceleration provided by modern graphics cards. The semantics of the library are designed to precisely match the specification of Render. Having the same semantics as Render allows for a seamless integration with the cairo library that then provides an attractive environment for developing new high performance graphical applications.

Hopefully, the work presented in this paper will be useful in the design of a new generation of hardware accelerated 2D graphics applications for X and the open source community in general.

## 1.3 The OpenGL Rendering Pipeline

To fully utilize the hardware acceleration provided by OpenGL, familiarity with the order of internal opera-

tions used in OpenGL implementations is of great importance. This is often visualized as a series of processing stages called the OpenGL rendering pipeline. This ordering is not a strict rule of how OpenGL is implemented but provides a reliable guide for predicting what OpenGL will do.

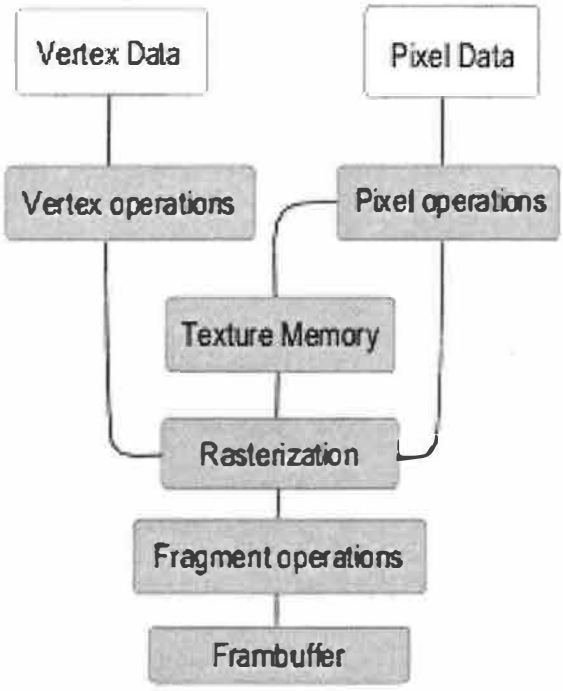



*Figure 2: The OpenGL Rendering Pipeline*

The latest generations of graphics hardware allow the application to replace the conventional vertex operations and fragment operations processing stages shown in figure 2, with application defined computations. These computations are defined with assembler-like languages that an OpenGL implementation compiles into vertex and fragment programs. The vertex and fragment programs provide an interface for directly accessing hardware and have been proven very useful in the development of Glitz.

## 2 Related Work

Some of the proprietary window systems have created their own graphics engines that can perform hardware accelerated rendering in a similar manner to the model discussed here. The one, that has probably attracted most attention is Apple's Quartz Extreme[5] compositing engine used in Mac OS X[6]. The user interface in Mac OS X is loaded with advanced graphics effects of the nature discussed in this paper. They all seem to run smoothly without bringing too much load on the CPU.

Microsoft is also developing something similar in their Avalon[2] graphics engine. It will be a fundamental part for hardware accelerated 2D graphics in the next windows version, currently being developed under the name Windows Longhorn[1].

Glitz is not the first Open Source graphics library that has been layered on top of OpenGL. An example, Evas[14]; a hardware accelerated canvas API, which is part of the Enlightenment Foundation Libraries. Glitz is unique compared to these libraries by using an immediate rendering model designed for the latest hardware extensions. Immediate data is resident in graphics hardware and off-screen drawing is a native part of the rendering model.

## 3 Implementation and Design

The development of Glitz and the other parts have been made in an entirely open fashion. Input from the open source community has been regarded.

### 3.1 Library Structure

As OpenGL layers are available for various platforms and systems, the library is designed to be usable with any of various OpenGL layers. Different OpenGL layers can be used by plugging them in to the core of Glitz through a virtualized backend system. Each backend needs to provide a table of OpenGL function pointers and few additional functions, which will allow for the core to perform its rendering operations unaware of the structure of the underlying OpenGL layer.

The core of Glitz is built as a separate library with a minimal set of dependencies. Each backend is built into its own library and different applications can use different backend libraries with the same core library.

The advantages of having a virtualized backend system and different backend libraries instead of just choosing the code to compile using preprocessor macros are important. It makes the link between the OpenGL implementation and the core of the library more flexible. It allows for the core of the library to be compiled for multiple backends.

As of now Glitz has two backends, for GLX[7] and AGL[4]. GLX is the OpenGL layer used on Unix[18] like systems to provide a glue layer between OpenGL and X. AGL is an OpenGL layer available in Mac OS. Backends for other OpenGL layers can be added in the future.

### 3.2 Rendering Model

The Render protocol describes an immediate rendering model. This means that the application itself maintains the data that describes the model. For example, with Render you draw objects by completely specifying what should be drawn. Render simply takes the data provided by the application and immediately draws the appropriate objects.

The opposite is to have a retained rendering model. A rendering model is operating in retained mode if it retains a copy of all the data describing a model. Retained mode rendering requires a completely specified model by passing data to the rendering system using predefined structures. The rendering system organizes the data internally, usually in a hierarchical database.

Principal advantages of immediate mode rendering includes a more flexible model and immediately available data that is not duplicated by the rendering system. However, it is more difficult to accelerate the immediate rendering model, because you generally need to specify the entire model to draw a single frame, whether or not the entire model has changed since the previous frame.

## 3.3 Off-screen Drawing

Off-screen drawing is an essential part of an immediate mode 2D graphics API. Support for off-screen drawing in OpenGL has been around for a long time on IRIX[8] systems and other workstations, but it is not until recently that it has become a standard feature on the regular home desktop computer.

Pixel buffers or so called pbuffers are what make off-screen rendering possible in OpenGL. Pbuffers are allocated independently of the frame-buffer and usually stored in video memory. The process of rendering to pbuffer is accelerated by hardware in the same way as rendering to the frame-buffer. However, as the pbuffer is a relatively new feature in the OpenGL world, it is not yet supported by all hardware and all drivers. When support for off-screen drawing is missing, the application using Glitz will have to handle this on its own. Even though Glitz is primarily designed for modern graphics hardware, it is important to be able to fall back on software rendering in cases where Glitz is not able to carry out off-screen drawing operations. For example, the cairo library handles this gracefully using image buffers.

## 3.4 User-Provided Immediate Data

As all representation of pixel data within Glitz reside in the graphics hardware's memory, application generated images must be transmitted to the hardware in some way. For this purpose Glitz provides two functions, one for transmitting pixel data to graphics hardware and one for retrieving pixel data from graphics hardware.

## 3.5 Image Compositing

The general composite operation is the fundamental part of the rendering model. It takes two source surfaces and one destination surface, where the second of the two source surfaces is the optional mask surface. Figure 3 illustrates this operation.

To composite one surface onto another with OpenGL, texturing of a rectangular polygon is used. This means

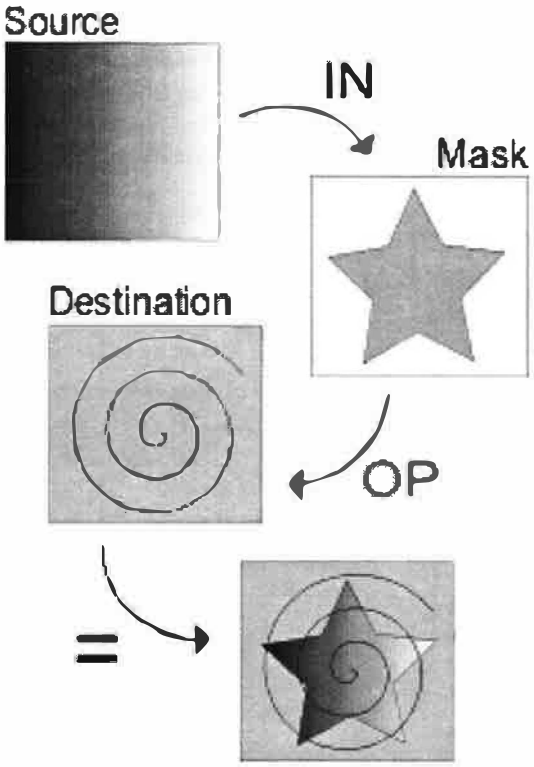



*Figure 3: src IN mask OP dst*

that the source surfaces must be available as textures. The default method for making a surface available as a texture is to have the graphics hardware copy the pixel data from a drawable into a texture image. As this is a relatively slow operation, Glitz does its best to minimize the area and number of occasions for this copy operation. On some hardware, a feature called render-texture is available that allows Glitz to completely eliminate the need for this copy operation and instead use a pbuffer directly as a texture.

The optional mask surface that can be provided to the general composite operation creates some additional complications. The source surfaces must first be composited onto the mask using the Porter-Duff in-operator and the result must then be composited onto the destination. The default method for handling this is to create an intermediate off-screen surface, which can be used for compositing using the in-operator. This surface can then be composited onto the destination with an arbitrary operator for the correct result.

The best way to do this would be to perform compositing with a mask surface directly without the creation of an intermediate surface. Even though the fixed OpenGL pipeline does not seem to allow for such an operation, Glitz is able to do this on hardware that support fragment programs. Fragment programs allow for fragment level programmability in OpenGL, and in combination

with multi-texturing, Glitz can perform composite operations with a mask surface very efficiently.

## 3.6 Image Transformations

Image transformation is a natural part of OpenGL and is efficiently done on all available hardware and with all available OpenGL implementations. Glitz transforms the vertex coordinates of the rectangular polygon used for texturing, and OpenGL will then in hardware handle fetching of correct source fragments for this polygon.

When using fragment programs for direct compositing with mask surfaces, some transformations cannot be done since the source surface and the mask surfaces share vertex coordinates. If this is the case, Glitz will be forced to not use direct compositing.

## 3.7 Repeating Patterns

To provide for solid colors and repeating patterns, surfaces have a 'repeat' attribute. When set, the surface is treated as if its width and height were infinite by tiling the contents of the surface along both axes.

Normally OpenGL only supports tiling of textures with dimensions that are power of two sized. If surface dimensions are of this size Glitz can let OpenGL handle the tiling for maximum efficiency. For surfaces that do not have power of two sized dimensions, Glitz will repeat the surfaces manually by performing multiple texturing operations.

Some OpenGL implementations support tiling of none power of two sized textures as well. If this is the case, Glitz will let OpenGL handle tiling of all surfaces.

## 3.8 Polygon Rendering

Glitz supports two separate primitive objects; triangles and trapezoids. Triangles are specified by locating their three vertices. Trapezoids are represented by two horizontal lines delimiting the top and bottom of the trapezoid, and two additional lines specified by arbitrary points. These primitives are designed to be used for rendering complex objects tessellated by higher level libraries.

Glitz only supports imprecise pixelization. Precise pixelization is not supported since OpenGL has relatively weak invariant requirements of pixelization. This is because of the desire for high-performance mixed software and hardware implementations. Glitz matches the following set of invariants for imprecise polygons.

- Precise matching of abutting edges
- Translational invariance
- Sharp edges
- Order independence

Hence the visual artifacts associated with polygon tessellation and translation are minimized.

### 3.8.1 Anti-aliasing

Aliasing is a general term used to describe the problems that may occur whenever an analog signal is point sampled to convert it into a digital signal, and the sample rate is to low. The number of samples do not contain enough information to represent the actual source signal. Instead the samples seem to represent a different signal of lower frequency, called an aliased signal.

In computer graphics, aliasing translates to the problems related to point sampling an analogous mathematical representation of an image into discrete pixel positions. With the currently available display devices it is simply not feasible to sample a non aliased signal, the resolution of the screen (the number of samples) is simply not high enough.

The results of aliasing are called artifacts. The most common artifacts in computer graphics include jagged profiles, disappearing or improperly rendered fine detail and disintegrating textures. The most obvious one, and the one that most applies to 2D graphics, is the jagged profile artifact. Figure 4 illustrates an aliased graphical image suffering from a jagged edge.

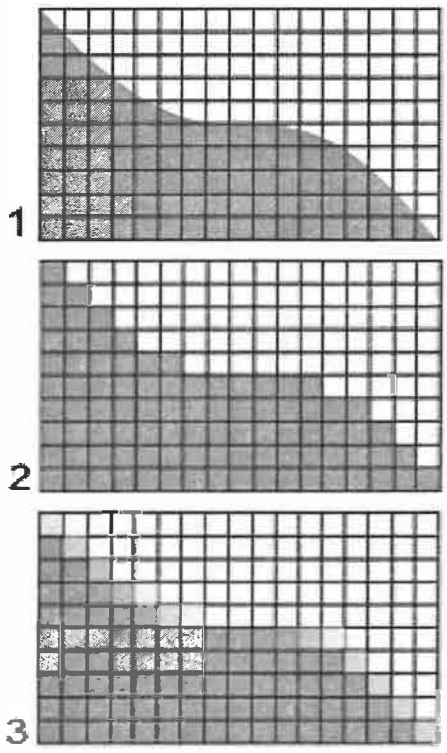



*Figure 4: (1) The mathematical representation of an edge (2) The edge, point sampled to screen pixels (3) The anti-aliased edge*

Anti-aliasing, naturally enough, is the name for techniques designed to reduce or eliminate this effect, usu-

ally by shading the pixels along the borders of graphical elements. There are several techniques that can be used to achieve anti-aliased graphics rendering with the OpenGL API. The most common techniques include:

- OpenGL's built in polygon smooth hint
- Multi-pass using accumulation buffering
- Multi-pass using blending
- Full-scene anti-aliasing using software super-sampling
- Full-scene anti-aliasing using hardware assist

All these anti-aliased drawing techniques are approximations. Each has its advantages and disadvantages. Different methods are suitable in different application contexts and have various support in graphics hardware and drivers. These methods have been investigated and evaluated with regards to performance and the result of the actual visual output. The challenge is to find a technique, or a combination of techniques, that will be able to provide nice anti-aliasing of the rendered scene in an as wide spectra of hardware and drivers as possible. As important as a nice on-screen result might be, performance issues are given a high priority in the search for a suitable anti-aliasing model. Another important criteria has been that they do not all fit well into the immediate rendering model used in Glitz.

The anti-aliasing model chosen for Glitz is flexible and other techniques can easily be added for special cases later on. The current implementation uses hardware assisted full-scene anti-aliasing.

This technique has been found suitable because it has a functional easy to use interface in OpenGL (through the multi-sample extension) and it fits well into Glitz without complicating the structure of the rendering model. It is relatively fast on current hardware and it produces adequate results in real-time rendering. The trend among graphics hardware manufacturers seems to be to favor multi-sampling over other anti-aliasing techniques in new products.

Full-scene anti-aliasing using hardware assist is typically implemented as multi-sampling, sometimes super-sampling. This is a very fast model that works for all primitives, interrelationships, and rendering models. It is also well supported in current hardware, since a couple of graphics card generations back. Some extra memory is required, but typically less than for software super-sampling or accumulation buffering. It yields decent-quality results but some people may not find them acceptable for small text. This does not affect the choice in this case however, as anti-aliasing of text will preferably be handled by an external font rendering library. On high end systems this technique has potential for generating extremely high quality results with a relatively low cost. Unfortunately, it is not always available for off-screen buffers (pbuffers).

The other techniques have been discarded mainly due to poor hardware support, high memory consumption, bad performance or poor results.

### 3.8.2 Indirect Polygons

Glitz has two different methods for rendering indirect polygons. Using an intermediate off-screen surface or using a stencil buffer.

The first method creates an off-screen surface containing only an alpha channel. The polygons are then rendered into this intermediate surface, which is used as mask when compositing the supplied source surface onto the destination surface. This method requires off-screen drawing support, and anti-aliased polygon edges can only be rendered if off-screen multi-sample support is available.

Whenever a stencil buffer is available, it will be used for drawing indirect polygons. The polygons are then rendered into the stencil buffer and the stencil buffer is used for clipping when compositing the supplied source surface onto the destination surface. This method for drawing indirect polygons is faster and does not require off-screen drawing support. When rendering to on-screen surfaces only on-screen multi-sample support is needed for anti-aliased polygons.

Indirect polygons can be used for pattern filling of complex objects.

### 3.8.3 Direct Polygons

Glitz is able to render polygons directly onto a destination surface. Each polygon vertex has a specific color associated with it and colors are linearly interpolated between the vertices. Direct polygons have the advantages of not requiring an intermediate off-screen surface or stencil buffer and are therefore faster, and supported on more hardware. Direct polygons might not produce the same results as indirect polygons when the alpha color component is not equal to one and should as a result not be used for complex objects with these properties. The more general indirect polygons should instead be used in these cases.

## 3.9 Text Rendering

Current version of Glitz has no built in text support. Glyph rasterization and glyph management could however be handled by the application or a higher level library. For efficient text rendering, glyph-sets with off-screen surfaces containing alpha masks, should be used. With external glyph management, Glitz renders text at approximately 50000 glyphs per second on the test setup described in section 4.

Built in text handling is planned for future versions of the library and tests have indicated that this should

increase glyph rendering speed to around 200000 glyphs per second.

## 3.10 Clipping

Render can restrict read and writes to a drawable using a clip-mask. Clients can create this clip-mask on their own or implicitly generate it using a set of rectangles. Glitz has a similar clipping interface but the clip-mask cannot be created by the application, it must always be implicitly generated from a set of rectangles, triangles and trapezoids. With Glitz, clipping only restricts writing to a surface. Glitz's clipping interface cannot restrict reading of a surface.

## 3.11 Programmatic Surfaces

Glitz allows you to create programmatic surfaces. A programmatic surface does not contain any actual image data, only a minimal set of attributes. These attributes describe how to calculate the color for each fragment of the surface.

Not containing any actual image data makes initialization time for programmatic surfaces very low. Having a low initialization time makes them ideal for representing solid colors.

Glitz also support programmatic surfaces that represent linear or radial transition vector patterns. A linear pattern defines two points, which form a transition vector. A radial gradient defines a center point and a radius, which together form a dynamic transition vector around the center point. The color of each fragment in these programmatic surfaces is fetched from a color range, using the fragments offset along the transition vector.

A color range is a one dimensional surface. The color range data is generated by the application and then transfered to graphics hardware where it can be used with linear and radial patterns. This allows applications to use linear and radial patterns for a wide range of shading effects. For example, linear color gradients and Gaussian shading. By setting the *extend* attribute of a color range to *pad, repeat* or *reflect*, the application can also control what should happen when patterns try to fetch color values outside of the color range.

Most programmatic surfaces are implemented using fragment programs and they will only be available on hardware supporting the fragment program extension.

Figure 5 shows the results from using programmatic surfaces for linear color gradients.

## 3.12 Convolution Filters

Convolutions can be used to perform many common image processing operations including sharpening, blurring, noise reduction, embossing and edge enhancement.

A convolution is a mathematical function that replaces each pixel by a weighted sum of its neighbors. The matrix defining the neighborhood of the pixel also specifies the weight assigned to each neighbor. This matrix is called the convolution kernel.

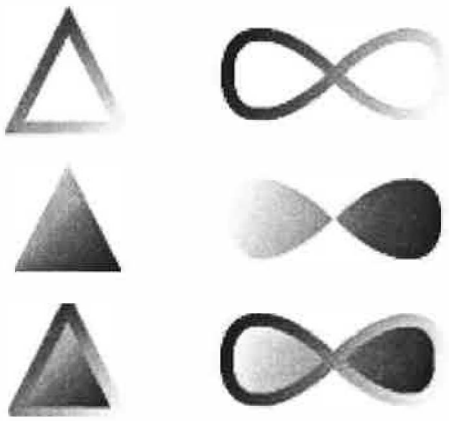


*Figure 5: Programmatic surfaces used for linear color gradients*

Glitz supports user defined convolution kernels. If a convolution kernel has been set for a surface, the convolution filter will be applied when the surface is used as a source in a compositing operation. The original source surface remains unmodified.

In Glitz, convolution filtering is implemented using fragment programs and is only available on hardware with fragment program support. The alternative would be to use OpenGL's imaging extension, which would require a transfer of all pixels through OpenGL's pixel pipeline to an intermediate texture. Even though the alternative method would be supported by older hardware, Glitz uses fragment programs in favor of performance.

This is an example of a convolution kernel representing a gaussian blur filter.

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

Figure 6 shows an image before and after applying a gaussian filter using the convolution kernel above.

## 3.13 A Cross-platform OpenGL Layer

Glitz's backend system works as an abstraction layer over the supported OpenGL layers and has genuine support for off-screen drawing.

In addition to the 2D drawing functions, Glitz also provides a set of functions that make it possible to use Glitz as a cross-platform OpenGL layer.

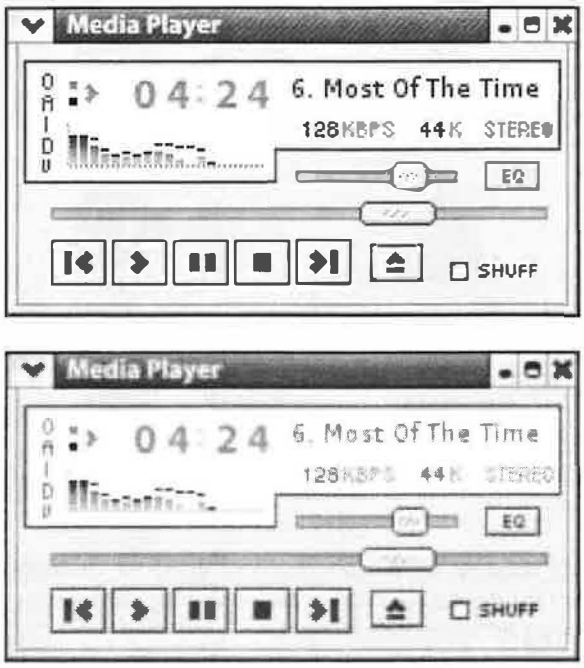



Figure 6: *An image before and after applying a Gaussian convolution filter*

The following three functions allow the application to use ordinary OpenGL calls to draw on any Glitz surface.

- glitz_gl_begin (surface)
- glitz_gl_end (surface)
- glitz_get_gl_texture (surface)

An application can initiate ordinary OpenGL rendering to a Glitz surface by calling the glitz_gl_begin function with the surface as parameter. The surface can be either an on- or off-screen surface. After a call to glitz_gl_begin, all OpenGL drawing will be routed to the Glitz surface. The glitz_gl_end function must be called before any other Glitz function can be used again.

An application can use both Glitz's 2D drawing functions and ordinary OpenGL calls on all surfaces as long as all OpenGL calls are made within the scope of glitz_gl_begin and glitz_gl_end.

glitz_get_gl_texture allows the application to retrieve a texture name for a Glitz surface. The application can use this texture name with ordinary OpenGL calls.

Figure 7 shows an example that render 2D graphics to an off-screen surface and then use it as a texture when drawing in 3D.

Applications, libraries and toolkits that use Glitz as rendering backend will get both 2D and 3D support with the ability two use all 2D surfaces as textures for 3D rendering.

```
offscreen_surface =
  glitz_glx_surface_create (display, screen,
                            GLITZ_STANDARD_ARGB32,
                            width, height);

glitz_fill_rectangle (GLITZ_OPERATOR_SRC,
                      offscreen_surface,
                      clear_color,
                      0, 0, width, height);

/* draw things to offscreen surface using
   glitz's 2D functions ... */

texture =
  glitz_get_gl_texture (offscreen_surface,
                        &name, &target,
                        &tex_width,
                        &tex_height);

glitz_gl_begin (onscreen_surface);

/* set up projection and modelview
   matrices ... */

glEnable (target);
glBindTexture (target, name);

glBegin (GL_QUADS);
glTexCoord2d (0.0, 0.0);
glVertex3d (-1.0, -1.0, 1.0);
glTexCoord2d (tex_width, 0.0);
glVertex3d (1.0, -1.0, 1.0);
glTexCoord2d (tex_width, tex_height);
glVertex3d (1.0, 1.0, 1.0);
glTexCoord2d (0.0, tex_height);
glVertex3d (-1.0, 1.0, 1.0);
glEnd ();

glitz_gl_end (onscreen_surface)

glitz_surface_destroy (offscreen_surface);
```

Figure 7: Rendering 3D graphics with a Glitz surface as texture

## 4 Results

Right now the library has not been tested that much in real applications since it is relatively early in the development process. Some test and benchmark utilities have been developed to analyze library functionality with respect to accuracy and performance.

### 4.1 Accuracy

The quality of the visual output from Glitz compares quite well to the corresponding output from XFree86's Render implementation (Xrender). Most objects tessellated by the cairo library are rendered without noticeable differences compared to Xrender using Glitz. However, in some complex objects a slight variation can be seen between Glitz's output and Xrender's output. Figure 8 and 9 illustrates the output inconsistencies for aliased rendering. The infinite sign is tessellated into 370 trapezoids by the cairo library, which are then rendered using Glitz and Xrender. Both figures are magnified to better show the output differences.

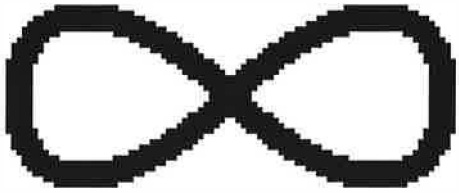


*Figure 8: Aliased OpenGL output*

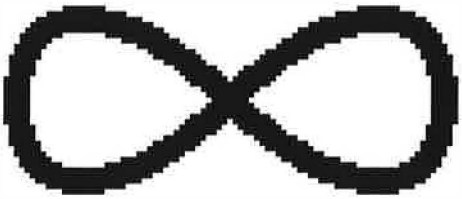


*Figure 9: Aliased xrender output*

Anti-aliased rendering may introduce some additional inconsistencies in the output between Glitz and Xrender. The number of samples used for multi-sampling has a big effect on the anti-aliasing quality. On older hardware, anti-aliasing is not even guaranteed, as it depends on relatively new OpenGL extensions. Nevertheless, if Glitz is run on fairly modern graphics hardware, very similar results are achieved with anti-aliased output. Figure 10 and 11 show the same infinite sign and illustrates the output inconsistencies for anti-aliased rendering.

Even though these results appear somewhat different in these magnified figures, the performance gained by using these OpenGL accelerated anti-aliasing techniques by far makes up for this. In most cases the generated

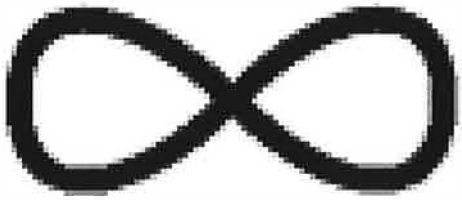


*Figure 10: Anti-Aliased OpenGL output (using 4 samples multi-sampling)*

*Figure 11: Anti-Aliased Xrender output*

results are close to indistinguishable.

### 4.2 Performance

A number of test and demo applications have been created during the development of Glitz to verify performance and functionality. This section presents results from a benchmark utility named rendermark. Rendermark compares the rendering performance of Glitz, Xrender and Imlib2[15] by doing a set of basic operations a repeated number of times. Comparison with Imlib2 is interesting as it is promoted as the fastest image compositing, rendering and manipulation library for X. Imlib2 performs all its rendering operations on client-side image buffers, so no on-screen rendering results are available for Imlib2.

Table 1 lists the tested output formats and table 2 shows the test setup.

| | |
|---|---|
| im-off | Imlib2 off-screen |
| xr-on | Xrender on-screen |
| xr-off | Xrender off-screen |
| gl-on | GL on-screen |
| gl-off | GL off-screen |

Table 1: *Rendermark Output Formats*

| | |
|---|---|
| CPU | 1GHz Pentium 3 |
| OS | Linux 2.4.22 |
| X11 | XFree86 4.3.0 |
| GPU | Nvidia GeForce FX-5600 (Nvidia's binary driver) |

Table 2: *Test Setup*

Each test is repeated a thousand times and the total time is shown in the tables.

### 4.2.1 Image Compositing

Image compositing performance is very important. For Xrender and Glitz this tests the composite primitive used for basically all rendering operations. Good performance here means good performance throughout the whole system.

- *over*: blends one image onto another using the over operator.
- *scale*. blends one image onto another using the over operator with additional scaling factors. Bilinear filtering is used.
- *blend*. blends one image onto another using the over operator with half opacity.
- *blur*: blends one image onto another using the over operator with a blur filter. For Glitz, this means applying a 3x3 mean blur convolution filter. The version of Xrender used for this test does not support convolution filters, and the test is therefore skipped.

Table 3 shows the image compositing results.

| | im-off | xr-on | xr-off | gl-on | gl-off |
|---|---|---|---|---|---|
| over | 3.809 | 3.870 | 3.850 | 0.109 | 0.107 |
| scale | 16.444 | 85.504 | 86.924 | 0.126 | 0.132 |
| blend | 4.349 | 73.222 | 69.613 | 0.264 | 0.263 |
| blur | 26.499 | - | - | 3.089 | 3.078 |

Table 3: *Seconds to complete composite test (lower is better)*

### 4.2.2 Color blend

This test evaluates color blend performance by drawing rectangles.

Table 4 shows the color blend results.

| | im-off | xr-on | xr-off | gl-on | gl-off |
|---|---|---|---|---|---|
| rect | 3.824 | 75.346 | 73.617 | 0.108 | 0.105 |

Table 4: *Seconds to complete color blend test (lower is better)*

### 4.2.3 Polygon Drawing

Polygon drawing is extensively used when rendering vector graphics. These tests show the performance when rendering the simplest polygon type, the triangle.

- *tri1*. Draws a set of triangles, each in a separate rendering operation.
- *tri2*. Draws a set of triangles in one single rendering operation. This type of operation is often used for rendering complex objects, which have been tessellated into (in this case) triangles. Imlib2 skips this test as it lacks support for it.

Table 5 shows polygon drawing results.

| | im-off | xr-on | xr-off | gl-on | gl-off |
|---|---|---|---|---|---|
| tri1 | 2.977 | 66.555 | 63.191 | 0.072 | 0.072 |
| tri2 | - | 2.078 | 1.950 | 0.030 | 0.030 |

Table 5: *Seconds to complete triangle drawing test (lower is better)*

### 4.2.4 Gradient Drawing

Tests linear color gradient performance. Xrender skips this test as it lacks support for it.

Table 6 shows gradient drawing results.

| | im-off | xr-on | xr-off | gl-on | gl-off |
|---|---|---|---|---|---|
| grad | 6.281 | - | - | 1.065 | 1.117 |

Table 6: *Seconds to complete gradient drawing test (lower is better)*

### 4.2.5 Hardware Accelerated Xrender

Nvidias's[10] binary XFree86 drivers contains an experimental feature that allows the driver to hardware accelerate the Render extension on XFree86's X server. Some Render operations are known to perform extremely good with this feature turned on.

| | xr-on | xr-off | gl-on | gl-off |
|---|---|---|---|---|
| over | 0.113 | 0.185 | 0.109 | 0.107 |
| scale | 84.659 | 86.217 | 0.126 | 0.132 |
| blend | 0.116 | 0.181 | 0.264 | 0.263 |
| blur | - | - | 3.089 | 3.078 |
| rect1 | 0.066 | 0.159 | 0.111 | 0.108 |
| rect2 | 0.070 | 0.228 | 0.113 | 0.118 |
| tri1 | 3.300 | 3.085 | 0.072 | 0.072 |
| tri2 | 1.597 | 1.688 | 0.030 | 0.030 |
| grad | - | - | 1.065 | 1.117 |

Table 7: *Seconds to complete test (lower is better)*

Table 7 shows that nvidia's driver performs well compared to Glitz in the cases where no transformations are used. In cases where transformations are used, Glitz is much faster than nvidia's driver, which most likely falls back on software rendering.

## 5 Conclusion

During the development of Glitz we have found that with the OpenGL API and the extensions available today, along with the wide range of hardware supporting them, a Render-like interface on top of OpenGL is viable and very efficient. This is an important conclusion as the desire for having an X server running on top of OpenGL grows rapidly.

The benchmark results points out Glitz's remarkable rendering performance. Even Imlib2's highly optimized rendering engine is no where near Glitz's performance.

Although performance is of high importance, the

greatest advantage with Glitz is that it provides a general way for accelerating the Render imaging model.

## 6 Future Work

Today, the existing implementation of the library supports all the basic functionality, which where initially set up for the project. However, there are still some important features missing, and the software is in an early stage of development with a lot of work remaining to make it stable and optimized with regards to performance and accuracy.

The following list contains those features that most importantly need to be addressed in future versions of the library.

- *Text rendering.* Built in text rendering will allow much higher glyph rendering speeds and remove complex glyph management from the application.
- *Sub-pixel rendering.* Sub-pixel rendering can be used to effectively increase the horizontal resolution of LCD displays. This will require support for compositing each color component with different masks.

The future will most certainly demand new features from the library, since it is an area of continuous development.

## 7 Visions

The X desktop seems to be going into a new era and cairo is definitely the 2D graphics API that will be used in tomorrow's X applications. The support for hardware accelerated surfaces in cairo might then be of great importance. Plans for the creation of an X server that will use OpenGL for all rendering are currently being made and this library, or the work behind the library, can hopefully be usable for this purpose.

## 8 Acknowledgments

We would like to thank Keith Packard, Carl Worth, and all of the people involved in the development of cairo for being helpful and encouraging. We would also like to thank our internal supervisor Berit Kvernes, along with the staff at the department of Computing Science at Umeå University, for supporting us in this project by approving it for financial funding in terms of study allowances.

## 9 Availability

All source code related to this project is free software currently distributed under the MIT license. The license of Glitz will follow that of cairo in case of changes.

The source can be retrieved via anonymous pserver access from the cairo CVS (`anoncvs@cvs.cairographics.org:/cvs/cairo`). The current status of Glitz and some additional information is available at `http://www.freedesktop.org/software/glitz`.

## References

[1] Microsoft Corporation. Online Resources: Windows Longhorn, December 2003. `http://msdn.microsoft.com/longhorn`.

[2] Microsoft Corporation. Online Resources: Avalon, April 2004. `http://msdn.microsoft.com/Longhorn/understanding/pillars/avalon/default%.aspx`.

[3] Adobe Systems Inc., editor. *PDF Reference: Version 1.4*. Addison-Wesley, 3rd edition, 2001.

[4] Apple Computer Inc. Online Resources: AGL, April 2004. `http://developer.apple.com/opengl/`.

[5] Apple Computers Inc. Online Resources: Quartz Extreme, Faster Graphics, December 2003. `http://www.apple.com/macosx/features/quartzextreme`.

[6] Apple Computers Inc. Online Resources: Mac OS X, April 2004. `http://www.apple.com/macosx`.

[7] Silicon Graphics Inc. Online Resources: GLX, April 2004. `http://www.sgi.com/software/opensource/glx/`.

[8] Silicon Graphics Inc. Online Resources: IRIX, April 2004. `http://www.sgi.com/software/irix/`.

[9] XFree86 Project Inc. XFree86: an open source X11-based desktop infrastructure, April 2004. `http://www.xfree86.org`.

[10] Online Resources: NVIDIA. Nvidia, April 2004. `http://www.nvidia.com`.

[11] Keith Packard. A New Rendering Model for X. In *FREENIX Track, 2000 Usenix Annual Technical Conference*, pages 279–284, San Diego, CA, June 2000. USENIX.

[12] Rob Pike. *draw - screen graphics*. Bell Laboratories, 2000. Plan 9 Manual Page Entry.

[13] Thomas Porter and Tom Duff. Compositing Digital Images. *Computer Graphics*, 18(3):253–259, July 1984.

[14] Rasterman. Online Resources: EVAS, April 2004. `http://enlightenment.org/pages/evas.html`.

[15] Rasterman and the imlib2 development team. Imlib2: An image processing library, December

2003. http://www.enlightenment.org/pages/imlib2.html.

[16] Robert W. Scheifler and James Gettys. *X Window System*. Digital Press, third edition, 1992.

[17] Mark Segal, Kurt Akeley, and Jon Leach (ed). *The OpenGL Graphics System: A Specification*. SGI, 1999.

[18] K. Thompson. Unix implementation. *The Bell System Technical Journal*, 57(6):1931–1946, July-August 1978.

[19] Mark Vojkovich and Marc Aurele La France. XAA.HOWTO. Technical report, The XFree86 Project Inc., 2000.

[20] Carl D. Worth and Keith Packard. Xr: Cross-device Rendering for Vector Graphics. 2003 ottawa linux symposium, July 2003.

# High Performance X Servers in the Kdrive Architecture

Eric Anholt
*LinuxFund*
anholt@FreeBSD.org

## Abstract

The common usage patterns of 2D drivers for the X Window System have changed over time. New extensions such as Render and Composite are creating new demands for 2D rendering which do not match those for previous architectures tailored to the core protocol. This paper describes changes made to the Kdrive X server implementation to implement new 2D acceleration, improve management of offscreen memory, implement OpenGL, and implement XVideo in a manner compatible with the Composite extension. With these changes, Kdrive is far better suited as a desktop X server than before and may serve as an example for desktop X server implementations. Simple benchmarks are presented.

## 1. Introduction

As desktop environments have advanced, the common usage patterns of the X Window System [1] have changed. In particular, the Render extension to the protocol has allowed for new graphical operations related to blending of images [2, 3, 4]. Software implementations of these operations are too slow for them to be used extensively in the user interface, and modern graphics hardware can provide dramatic performance improvements for many operations. The new Composite extension is increasing the amount of rendering to offscreen memory, and creates new requirements for rendering of older operations. At the same time, rendering operations provided by the core protocol are being used less frequently. Improving performance of these new operations will enable significantly greater use and permit a new user experience for the Linux desktop. This paper will explore some of the changes made to the Kdrive-based X server (also known as TinyX or Xkdrive), to improve performance and create a server suitable for the Linux desktop. This introduction will describe the basic components of an X server, the features of hardware typically found in desktop computers, and describe some of the features and requirements of some X extensions related to the work here. Section 2 will describe another X server implementation, XFree86, and how it compared to Kdrive. Section 3 will cover the changes made to Kdrive as a result of research into making Kdrive-based X servers useful for the desktop. Section 4 will cover some of the tangible results of this work, and Section 5 will describe some of the changes remaining to be made to Kdrive to make it into a capable desktop X server.

### 1.2. X Server Architecture

The X sample server developed for the MIT X Consortium by Digital Equipment Corporation in 1987 provides a large base of shared code. Most X servers available today, including the two mentioned in this paper – XFree86 and Kdrive, take advantage of this. This shared code includes parts for handling basic X server initialization and teardown, for decoding the network protocol, for breaking rendering requests into simpler operations, for performing rendering operations to memory. It also includes extension implementations, among other things. Different X servers using this shared code base are implemented in the Device Dependent X (DDX) part of the server, located under the *hw* directory in the source tree of both XFree86 and Kdrive. The DDX is responsible for actually dealing with input and output, whether that is directly to hardware (in the case of XFree86 and Kdrive) or to some software layer (as in the xnest or the virtual framebuffer servers located in the XFree86 source tree).

### 1.3. Features of Commodity Hardware

Most video cards in desktop computers support the following 2D acceleration functions:

- Filling rectangles with a solid color
- Copying one area of the screen to another
- Drawing lines using the Bresenham algorithm
- Performing color expansion of monochrome bitmaps

Color expansion is the process of filling pixels of the screen with foreground/background colors based on a monochrome bitmap, which is used for non-antialiased text rendering.

Typically, each of these operations can be done with a Raster Operation (ROP) that allows various binary operations to be done using the source or foreground/background colors and the destination. A planemask can also be set that allows writing to only certain bits of each pixel.

Most cards also have the ability to take a video image (which is stored in a YUV format that requires conversion before it can be displayed on an RGB display) and scale it onto the screen in places where the color on the screen matches a color key.

The core X protocol supported these 2D capabilities well, but with the advent of 3D hardware, new possibilities are opened up for 2D rendering. By rendering rectangles using 3D hardware designed for OpenGL [5] or DirectX, we can perform blending of source images into destination images, allowing for fast rendering of effects like transparent windows, shadows, and the operations necessary for antialiased text. A card supporting 3D typically has one or two texture units, but sometimes more. Each texture unit can take input from a source image in card memory (also referred to as offscreen memory or framebuffer) or AGP memory (system memory accessible directly by the card) when performing rendering. These textures can be of varying color depths or even non-RGB formats like YUV, can include alpha channels for controlling blending, and can be scaled and mixed with each other in various ways to produce a color value to be written to the destination. With the final pixel value, various blending functions can be used based on its alpha value and the alpha value of the pixel currently in the destination, which typically map to a set of blending functions provided by OpenGL.

## 1.4. Render

The Render extension provides several new rendering operations in X servers to handle the new demands of desktop applications, designed to target the 3D capabilities of newer hardware. It begins by creating a new type, the Picture, which wraps around an X drawable (a pixmap or a window). A Picture adds a fourth color channel, alpha, which indicates the amount of opacity. The Picture also adds the ability to apply transformations to the coordinates of pixels requested from the drawable and to make coordinates outside of the drawable wrap around ("repeat"). The new rendering commands that can be performed on Pictures include:

- blending of a constant color into a destination rectangle
- blending of rectangles from images into the destination
- blending of triangles from images into the destination
- blending of trapezoids from images into the destination
- blending of a series of glyphs into the destination

In the sample implementation of the Render extension used by both XFree86 and Kdrive, all of those graphical operations are implemented using the same basic composite operation for blending of rectangles. This call ("Composite") takes the following arguments:

- a source picture and starting x/y coordinates
- an optional mask picture and starting x/y coordinates
- a destination picture and starting x/y coordinates
- the width and height of the rectangle to be rendered into the destination
- a composite operation

For each pixel in the destination rectangle, a source pixel is chosen based on the offset within the destination currently being rendered, after modification by the transform and repeat requirements of the source picture structure. If a mask is included, the source pixel is multiplied by the alpha value of a pixel chosen similarly from the mask picture (unless component alpha is being used, to be discussed in the "Complications for Render Acceleration" section). This calculated source pixel value is then blended into the destination pixel according to the composite operation chosen, as described in *Design and Implementation of the X Rendering Extension* by Keith Packard [2].

## 1.5. Composite

The Composite extension (not to be confused with Render's Composite operation) is an X server extension that was first implemented in the Kdrive X server. It

allows an X client to control the compositing of a window hierarchy into a parent window. It does this by redirecting the rendering of that window hierarchy (or, to simplify the discussion, a single window) into an offscreen pixmap. This X client (a compositing manager) can then use the Damage extension to be notified when an area of the window changes, and draw the appropriate area in the parent window as necessary. Because the client controls the drawing into the destination, it can add effects such as translucency, shadows, or reshaping of windows. Composite can also be used to simply get at the pixels of a window that would otherwise be obscured. The first publicly available compositing manager was xcompmgr, which adds shadows to windows and provides translucent menus. The use of xcompmgr also eliminates the delay between a window being exposed (for example after a window on top of it is moved) and the redraw of the window underneath. This provides a smoother user experience, with less of the flicker associated with moving and resizing of windows.

### 1.6. XVideo

The XVideo extension is used to perform conversion and scaling of video formats such as YUV in the X server. These conversions can be very expensive if done in software, and they are the primary time consumer in video playback.

### 1.7. OpenGL

OpenGL support is necessary for X servers due to its widespread use in games and other applications on the desktop. The GLX extension allows sending OpenGL commands over the X protocol to be rendered by an X server. Because the wrapping of OpenGL commands into GLX requests creates significant overhead, it is desirable to avoid that process. To avoid the overhead, direct rendering is implemented to give the client direct access to the video card. However, direct rendering introduces security risks because many cards can issue DMA requests to read or write system memory, and thus access has to be limited. Direct rendering also requires synchronization of access to the hardware as well as kernel assistance in submitting DMA requests and handling interrupts.

## 2. Related Work

XFree86 is the most popular open-source X server, and is the de facto standard for comparison of other X servers. This section describes significant differences between XFree86 and Kdrive.

### 2.1. XAA Versus KAA

Part of the DDX is responsible for handling hardware acceleration of drawing. In XFree86 and Kdrive this piece is called the XFree86 Acceleration Architecture (XAA) or Kdrive Acceleration Architecture (KAA) [6]. The design decisions behind the two implementations vary greatly.

XAA manages offscreen memory by treating it as a large 2D area at a set number of bits per pixel (bpp). Pixmaps (offscreen images) of any other bpp therefore cannot be stored in offscreen memory. Acceleration hooks are implemented as two or more callbacks to the driver. One is a Setup hook which sets up the hardware for the planemask and ROP, and performs other preparation for rendering that operation. The Subsequent hook can then be called one or more times to actually perform rendering. This division is an advantage because it avoids repeated Setup calls in the case where clipping results in the operation needing to be done in separate pieces. The video card is set up so that the visible screen (offset 0) is the source and destination offset, the screen's width is the pitch, and offscreen areas are simply areas with "y" values beyond the height of the screen. The Setup hook is not allowed to fail, so many flags are available for drivers to tell XAA about the hardware's features, such as planemask and transparency support or limitations on the direction of screen-to-screen copies. Also, accelerators have limits on coordinates for 2D acceleration, so often the memory for offscreen pixmaps has to be limited so that offscreen areas with "y" values beyond the limits are not used. XAA has hooks for copying areas, filling rectangles and trapezoids, drawing different types of lines, color expansion of monochrome bitmaps, uploading images from system memory to screen memory, and Composite operations from memory (described in Section 2.2).

In contrast, in KAA the offscreen memory is managed as a linear area. This allows pixmaps of different bpp to be stored offscreen, which is a requirement for fast acceleration of Render operations, where using pixmaps of different bpp from the screen is the norm. Because the offscreen memory cannot be represented as a single 2D area, the pixmaps have to be passed in to

the acceleration hooks, and the drivers then set up the hardware's offset and pitch registers for the source and destination each time. This has the added benefit that there are not the same limitations on the total offscreen memory due to the size of the hardware's 2D coordinates. To avoid the complexity of flags for KAA to determine whether an operation can be accelerated, the Prepare hook (equivalent to Setup in XFree86) is allowed to fail, which tells KAA to fall back to software rendering. An additional Done hook is included to allow for any cleanup necessary after a rendering operation, but the Done hook is a stub in most drivers. KAA implemented only two types of acceleration before the work for this paper, which were screen-to-screen copies and solid fills. It was decided that the other types of acceleration supported in XAA, such as lines, trapezoids and color expansion were used rarely enough in modern desktops that they did not justify the additional complexity to accelerate.

## 2.2. XAA Implementation of Render Acceleration

XAA implements a small subset of Render operations. It handles only cases where the destination picture is located in offscreen memory and the source picture is not. Because XAA's 2D offscreen memory layout prevents the offscreen storage of pixmaps with a bpp different from the screen, many source images will be located in system memory anyway. This means that typically the source picture has to be uploaded into a scratch area each time before the hardware can perform acceleration from it. Furthermore, the only case where composite with a mask is supported is when the source is a 1x1 repeating picture (a solid color). The hooks for this acceleration of compositing are implemented for Matrox Gx00 cards in the mga driver, the vmware emulator in the vmware driver, and possibly for SiS 300-series cards in the sis driver.

## 2.3. OpenGL in XFree86

In XFree86, OpenGL is implemented using the Direct Rendering Infrastructure (DRI). The DRI consists of several components: a kernel module specific to the video card ("Direct Rendering Module" or DRM), a DRI-aware 2D driver in the DDX, the GLX extension, the XF86DRI extension to the X protocol (used for communicating information about the DRM and hardware setup to the client), and the 3D driver itself, which is a card-specific shared library opened by the OpenGL library (libGL) .

The kernel module is used by the X server to set up a shared memory area that contains information about the video card. That area includes a lock, which is used cooperatively by clients and the server to arbitrate access to the card, along with card-specific state that is managed by the clients and server. Though clients could misbehave and abuse the lock, this can at worst lead to a lockup of the hardware and should not allow a security compromise. The DRM also often allows allocation of DMA buffers and submission of sets of commands by DMA, waiting for hardware interrupts, and other interactions with the hardware which cannot be handled or are difficult to handle in userspace.

Because of the availability of source to the DRI drivers in XFree86, many of the drivers are very similar in their structure. Most drivers allocate memory for OpenGL's back and depth buffers statically at server startup or when the first 3D client is started. Clipping is used so that all clients can share the back and depth buffers and only render to the same rectangles in the back buffer as they would on the visible screen. Because the design decision to use these static buffers was made originally, pbuffers (pixel buffers not associated with the screen) are not implemented, though they are in demand. The SiS300 driver is different in that it allocates back and depth buffers per client from a static block of card memory managed by the kernel module. However, its design is not a good model for other drivers because it cannot handle failure to allocate memory for buffers or textures.

In XFree86, all indirect rendering (OpenGL commands sent through GLX) is performed in software. Although the ability to hardware-accelerate indirect rendering is desired, it has not been implemented yet. The primary reason is that the interface between the server GLX code and the current software rendering core ("libGLcore") is very different from that between libGL and a DRI 3D driver.

## 2.3. XVideo in XFree86

In XFree86, XVideo is almost always implemented using the card's overlay scaler. With this method, the window where the video is to be shown is painted with a color key. The video image is loaded into offscreen memory, and the hardware is set up to paint the image appropriately into the card's video output where the color key matches. Most, or most likely all, hardware has only a single overlay scaler, meaning only one video can be played at once.

One exception is the driver for the Matrox Gx00 series hardware, which offers textured video as an option along with the overlay scaler video, though enabling textured video is exclusive of the DRI and overlay video. With textured video, the 3D hardware is used to perform the conversion and scaling of the video, and supports multiple ports (many videos being displayed at once).

## 3. Acceleration in Modern X Servers

In the past, Kdrive has been targeted for X server development and not for general desktop use. The changes researched for this paper were oriented towards making Kdrive an option for a desktop server by implementing new acceleration and making an example of acceleration for other servers to follow. This consisted of creating a better architecture for Render acceleration, implementing GLX in the server, improving management of offscreen memory, and implementing XVideo in a manner more appropriate for the Composite extension.

### 3.1. Complications for Render Acceleration

The Composite operation implemented by Render is complicated, and it cannot be implemented in hardware in all cases. Having two source images for an operation (source and mask) typically requires the use of the 3D hardware to implement. Most hardware cannot support 8-bit alpha images (which are very common) for source or destination when blending based on alpha values is necessary. Most older hardware 3D engines cannot handle textures with a width or height that are not a power of two number of pixels (NPOT), or are unable to do repeat (wrapping of texture coordinates outside the boundaries) for NPOT textures. Unfortunately, NPOT sources are the norm in 2D operations. Also, the alpha component of source, mask, or destination can be located in a separate buffer from the color data, which will be difficult to implement in hardware. Often the width or height of the source or destination can be larger than the hardware's limits on texture sizes (2048 pixels being common in newer hardware, but as low as 256 on older hardware).

Finally, component alpha will be difficult to accelerate. Component alpha rendering is used most frequently for sub-pixel rendering of anti-aliased fonts, taking advantage of the known order of the red, green, and blue bands in LCD pixels to provide additional screen resolution. When component alpha is used, the mask value is actually a set of four alpha values instead of one. Each source channel is multiplied by the corresponding mask channel to produce the final source color value, and each mask channel is multiplied by the source alpha to produce the final source alpha value. Then, each of these source value and source alpha pairs is blended into the appropriate destination channel using the specified composite operation. Although it should be possible to produce the correct source color values, current hardware can only do the alpha blending stage using a single source alpha value and not the componentized source alpha required. Composite operations that involve the source alpha value will most likely require using a multiple-stage process to accelerate, which may be difficult to implement.

### 3.2. Implementing Render Acceleration in Kdrive

There are two tasks for implementing Render acceleration. One is to accelerate Composite using existing 2D acceleration hooks whenever possible. The other is to create new hooks for common operations that map well to hardware features. This section covers some of the work on accelerating the most common operations desired by current applications.

Originally, KAA only implemented acceleration for Composite for CopyArea equivalents. This is when the operation is "Src" and there is no repeat flag set, no transform of source coordinates, and no mask. The xcompmgr application uses this operation very frequently to copy windows to the backbuffer of the screen, and to copy the backbuffer to the screen. The acceleration was implemented by using the standard Copy hook that all drivers had to implement.

The first new Composite acceleration hook implemented in Kdrive is called "Blend." Blend is a set of three hooks for a Kdrive driver: PrepareBlend, Blend, and DoneBlend. This mirrors the three hooks each for the Copy and Solid operations already implemented in Kdrive. In the Blend case, there is a source image but no mask image, and both source and destination are located in framebuffer memory. This may be possible to accelerate using only the "front-end scaler" of a card, originally targeted for video scaling, or also with the 3D hardware of cards with a single texture unit. In the PrepareBlend stage, the hardware is set up for the pictures and composite operation that are

passed in, and failure can be returned to signal that software fallback is necessary. The Blend call takes the source and destination coordinates and height and width of a rectangle to be blended, with no option to fail. DoneBlend is called after a set of Blends, in case some teardown is necessary. Blend was first implemented for ATI Rage 128 (R128) hardware, but has since been disabled due to problems found using a Render extension test program.

The second new acceleration for Composite is to check for cases where the source is a 1x1 repeating picture and there is no mask, with the "Op" operation being "Src." This can be accelerated using the solid-fill driver hook, which is also required for Kdrive drivers. The source is converted in software into the X server's Pixel type and passed to the Solid hook. In contrast to other hardware acceleration, this hook prefers that the source be in memory rather than the framebuffer, because if it is located in the framebuffer the accelerator has to be idled before the value can be accessed by the CPU for conversion to the destination's pixel format.

The next new acceleration hook provided is a set of PrepareComposite/Composite/DoneComposite calls, where the only thing checked by KAA is that the source, destination, and the optional mask are located in framebuffer memory. The driver is responsible for checking everything else, including handling transform, repeat, and all the various formats. This is useful for newer hardware like the Radeon where almost all commonly-used Composite operations can be implemented in the 3D hardware. The Composite acceleration was first implemented for ATI Radeon 100-series hardware, and subsequently for ATI Rage 128 hardware.

Finally, a new UploadToScratch hook was added that takes two pixmap pointers and makes the second a copy of the first, but with the data located in a scratch area in card memory. This allows temporary migration of a pixmap for the case where a pixmap not being in offscreen memory is all that is preventing acceleration. This occurs frequently in drawing of glyphs, which are not stored in real pixmaps and therefore will not be migrated into offscreen memory normally. The previous allocation to the scratch area is invalid whenever a new UploadToScratch call occurs.

## 3.3. Offscreen Memory Management

Properly managing offscreen memory is a critical feature for an X server, and Render operations that read from the destination are making the problem more visible. This is because video cards typically have very slow framebuffer read speeds, making software fallbacks that result in the CPU reading from card memory very expensive. A test on the Rage 128 showed a 23% slowdown when writing to framebuffer compared to system memory, versus a 99% slowdown when reading from the framebuffer instead of memory, as seen in Table 1.

| | ***read*** | ***write*** |
|---|---|---|
| R128 system | 531MB/sec | 247MB/sec |
| R128 AGP | 14.4MB/sec | 443MB/sec |
| R128 FB | 5.11MB/sec | 192MB/sec |
| Trident system | 228MB/sec | 160MB/sec |
| Trident FB | 9.74MB/sec | 15.9MB/sec |

*Table 1: Read and write speed to 512KB blocks of system, AGP, and framebuffer memory on a 700 Mhz Pentium 3 with ATI Rage 128 Mobility M4 and a 300Mhz Pentium 2 with Trident Cyber 9525/DVD.*

The original Kdrive offscreen memory management system was very simple. A score is kept that marks whether a pixmap should be in framebuffer or system memory. Pixmaps with a size larger than a certain value (some heuristic) are moved into offscreen memory when allocated. When a copy operation occurs, the source pixmap has its score increased if the destination is in offscreen memory, or decreased if it is not. When an unaccelerated Composite operation occurs, the source and mask have their scores decreased, and when an accelerated Composite operation occurs their scores are increased. Other operations performed in software, such as core protocol text rendering, do not modify the score. If the score reaches the *move in* threshold, it is moved into offscreen memory if possible, replacing other pixmaps as necessary to do so. The replacement process starts at the beginning of offscreen memory and continues until enough space is freed (approximately -- it does not start moving out pixmaps until it locates a sufficient stretch without hitting locked offscreen areas). If the score goes below the *move out* threshold, the pixmap is moved back from offscreen memory to system memory. The scores are clamped to a minimum and maximum to make the migration more responsive to changing usage of a pixmap. The score values can be seen in Table 2, and it is not clear from

CVS logs if the values are tuned or simply based on estimates.

| *Score name* | *Value* |
|---|---|
| KAA_PIXMAP_SCORE_MAX | 20 |
| KAA_PIXMAP_SCORE_MOVE_IN | 10 |
| KAA_PIXMAP_SCORE_INITIAL | 0 |
| KAA_PIXMAP_SCORE_MOVE_OUT | -10 |
| KAA_PIXMAP_SCORE_MIN | -20 |

*Table 2: KAA pixmap migration thresholds.*

This initial memory management system worked well enough for the initial goal of getting hardware acceleration between pixmaps and to the screen, but it has limitations. It does not migrate destination pixmaps that are being software-rendered towards system memory, which includes some significant core operations like text rendering and PutImage. It also has problems with thrashing. As soon as more pixmaps should be offscreen than there is sufficient memory for, choosing which pixmaps to keep in memory becomes an issue. First, when a pixmap is replaced by another allocation, it will not be moved back into offscreen memory until its score goes below the *move in* threshold and back above it again. If the *move in* process is changed to move any pixmap back in that has a score above the *move in* threshold, thrashing occurs when the first pixmaps in offscreen memory are chosen to be replaced each time. This state can be seen after anywhere from seconds to a few minutes of typical desktop usage.

To fix this, several things were changed. The first improvement was to modify the pixmap migration score to express whether there is more overall need for that pixmap to be in framebuffer or system memory, rather than how often it is used as a source image in a screen to screen copy to framebuffer versus system memory. This means that acceleration operations like Copy and the various solid operations were made to migrate both source and destination toward framebuffer. Also, the KdCheck* software fallbacks that are not called as a result of a failure of one of the acceleration operations were made to migrate the destination toward system memory.

The next change was to keep a new score in the structure for each allocated offscreen area. Unlike the migration score kept in the pixmap private structure, this one is meant to represent how important it is for a particular offscreen area to stay there. This new score is increased by a constant value whenever a pixmap gets its migration score increased. Periodically the scores of each offscreen area are reduced by a fraction so that the accumulated scores of offscreen areas decay over time. When a new area is to be allocated, the set of offscreen areas with the lowest total score is chosen to be replaced, rather than simply the first set of areas found. This system is not optimal, because a pixmap that gets replaced loses all of its accumulated score. Also, the score is not a very accurate representation of the value of a particular offscreen area being offscreen. However, it is sufficient to reduce the thrashing problem that existed.

The final change was to add a "dirty" flag to the pixmap private structure. Whenever the pixmap is modified, the dirty flag is marked, and it is cleared when the pixmap is moved into or out of offscreen memory. If the dirty flag is not set when a pixmap is to be moved out of offscreen memory, the expensive process of reading the pixmap from the framebuffer is skipped.

One failed experiment was based on the difference between the framebuffer and system memory speeds on the Rage 128. That experiment was to throw out clean offscreen areas when the migration score is decreased (when the pixmap is about to be used for software rendering), along with the usual move out process of pixmaps whose score goes below the threshold. However, the process of moving pixmaps in and out appears to outweigh the advantage of avoiding some software rendering to the framebuffer.

## 3.4. Issues With Direct Rendering in Kdrive

The Composite extension creates several challenges for implementing the DRI. The existing DRI drivers which we would like to use are designed only for a static front/back buffer and a front buffer which is actually visible onscreen. With the Composite extension, however, rendering may need to target a dynamically allocated buffer located in offscreen memory. Therefore the kernel's knowledge of the front/back buffer location (if it has any) must be per graphics context rather than per-device. Also, for a direct rendering context, the server must be notified in some way when the client updates its front buffer, so that damage can be computed.

## 3.5. Implementation of OpenGL in Kdrive

At the time of this writing, the work to implement hardware-accelerated OpenGL in Kdrive is incomplete. The first step was to bring in a software implementation of GLX. The GLX code was taken from XFree86 (actually DRI CVS, a separate repository for OpenGL development in XFree86), and the server build was modified to use a CVS checkout of Mesa (rather than including a copy of Mesa source code in the server tree as in XFree86). At this time, it is limited to only functioning when Composite is disabled, but it allows the XFree86 libGL to function.

Next, the server-side component of the DRI was brought in from XFree86 and the ATI driver was modified to initialize the DRM and submit its 2D commands through DMA. All that remains to get direct rendering in Kdrive on par with XFree86 appears to be debugging the ATI driver's initialization of the DRM.

### 3.6. XVideo

Implementing XVideo with the overlay scaler has several limitations. One limitation is that there is usually only one overlay scaler port, so only one video can be handled at once. It also prevents capturing screenshots, because the screenshot will include the color key instead of the scaled, converted video. Finally, the presence of the Composite Extension means that blending may occur over the video window's contents, so that the color key will not match and a blended color key will appear instead of the blended video.

The solution is to use the 3D hardware or front-end scaler to do the conversion and scaling of YUV data for XVideo. The Rage 128 and Mach64 have front-end scalers. The ATI Radeon and Rage 128, Matrox Gx00-series, and 3dfx Voodoo3+ should all support YUV data as textures (though the 3dfx older than the Voodoo4 may not be useful due to limitations on texture sizes). Video using the front-end scaler was implemented on the Rage 128. However, one problem with doing video scaling using the texture units is that there are fewer controls of the output. The overlay scaler typically has controls for brightness and saturation, while the Rage 128 front-end scaler and texture capabilities appear to only have a temperature (whitepoint) control and no brightness/saturation. Lack of brightness or saturation control may be possible to work around by combining the video with a secondary texture.

## 4. Results

The Render acceleration is by far the most measurable acceleration implemented as a result of this work. Even without support for all the operations necessary, the Composite acceleration on Radeon 100-series hardware tripled the speed of non-subpixel antialiased text rendering, according to x11perf's aa24text test (2x533Mhz Celeron, Radeon 7500). The Rage 128 Composite implementation showed the largest improvement, with over a fivefold improvement in non-subpixel antialiased text rendering, as seen in Table 3. It also greatly improved the perceived speed of using xcompmgr, which frequently uses Composite Over operations with a 1x1 repeating mask picture.

| | *aa24text* | *rgb24text* |
|---|---|---|
| R128 software | 14200/sec | 11900/sec |
| R128 hardware | 78500/sec | 2550/sec |

*Table 3: x11perf results for antialiased text (aa24text) and subpixel-rendered antialiased text (rgb24text) on a 700Mhz Pentium 3 with Rage 128 Mobility M4, before and after hardware acceleration of Composite.*

However, at the same time as non-subpixel text rendering speed increased, a near fivefold decrease in subpixel rendering speed was seen. This is because at least some of the operations necessary, particularly component alpha blending, are not supported by the Rage 128 Composite implementation, so the effort that goes into migrating the pixmaps (including UploadToScratch) to make them possible to accelerate is wasted. This suggests more offscreen memory management work is needed, to avoid that extra effort when it is not necessary.

## 5. Future Work

At this point, it needs to be decided if Kdrive is an appropriate server architecture for use as a desktop server. In particular, it lacks drivers for most hardware and lacks the support for control of video modes that XFree86 offers, though it now offers better 2D acceleration for Render operations and improved offscreen memory management. This section will describe future work to be done on 2D acceleration based on the new work in the previous section, and the OpenGL work necessary to make Kdrive usable as a desktop server.

## 5.1. Render Acceleration

Even after adding the catch-all Composite hook, hooks for simpler operations such as Blend remain useful because they can be tailored to common hardware features. These hooks free driver authors from having to figure out the exact set of conditions under which they can accelerate, and from needing to manually migrate and pull out the data.

There is at least one more Render acceleration hook that could be useful. One common operation is to do a Composite with an ARGB source and a 1x1 repeating mask to blend opaque images over the destination with a constant alpha value. This could be done on hardware with only one texture unit or with a front-end scaler by putting the constant mask value in the hardware's primitive color registers instead of using a secondary texture map. KAA would synchronize the accelerator if necessary, pull the value from the mask picture, and pass it, along with the same arguments as Blend, to a new BlendMask hook.

## 5.2. Offscreen Memory Management

There are still many things that could be done to improve offscreen memory management. There are circumstances where a pixmap should be marked for migration out of framebuffer which are not being covered. However, the current system for measuring whether a pixmap should be in framebuffer is very crude, being based on the number of software versus hardware operations, rather than any measurement of the cost of those operations.

Also, there are serious weaknesses in how migration is being dealt with for operations that are going to fail (such as component-alpha compositing, or unsupported composite operations). Currently, we do not migrate operations failing in the Setup stage away from the framebuffer, which is a major speed penalty. If we do migrate failing operations away, we'll see flip-flopping of the migration. This is because for repeated rendering of the same failing operation, the pixmap will always moved toward the framebuffer until it is migrated, at which point the Prepare will start failing and it will start to be moved back toward system memory. The solution will most likely require adding a fourth hook that performs all of the checking of the operation before any migration towards framebuffer happens and changing the Setup hook to be non-failing.

The excellent write speeds to AGP memory suggest that it might be a good choice to use for the scratch area for the UploadToScratch hook instead of framebuffer when available, if the penalty for the card to read from AGP instead of local memory is not too heavy.

If accessing AGP memory from the card does not include much overhead, it may be valuable to set up hardware with limited offscreen memory to have a large piece of AGP memory addressable and use that as an extension of the card's local memory.

Another area to research would be working on a method to calculate an optimal working set of pixmaps to be located in offscreen memory, based on how often they are dirtied, how often they get read from and written to, size, and other factors. This could be a significant improvement over the current system of simply moving in whatever is necessary for the current operation. Designing a complete solution to this may be hard, and benchmarking varying implementations is difficult due to the lack of a standard general desktop usage benchmark to be run that would stress offscreen memory management.

## 5.3. OpenGL

The next step in developing quality OpenGL support is to fix the problems with Composite in the GLX implementation. When that is completed, the major piece of work is to convert the software GLX implementation to use a software-rendering DRI driver. This would basically be a wrapper around the same sort of software GLX implementation, but with a normal DRI driver interface. Once that is completed, it should be straightforward to replace the software-rendering DRI driver with a hardware-rendering DRI driver, thanks to work that has been done in Mesa CVS to produce DRI drivers that do not rely on the X protocol. The drivers should soon be ready for rendering to buffers other than statically allocated back/depth buffers because of work being done to support pbuffers in the DRI and Mesa projects.

## 5.3. XVideo

Despite being an improvement visually, the current XVideo acceleration is still a large consumer of CPU time when playing movies. The CPU usage may be attributable to the copying of data to the framebuffer,

in which case using AGP memory when available may help.

Also, most overlay scaler implementations of XVideo allow control of saturation and brightness, while the current R128 XVideo does not offer these controls. More work needs to be done to see if these controls can be made for the textured video, either using the same scaler setup as for the overlay scaler, or using texturing features.

## 6. Availability

All work described here is available under the MIT/X11 license, and instructions for getting the code are available at:

http://www.freedesktop.org/~anholt/freenix2004/

## 7. Acknowledgments

Many thanks to Keith Packard for his guidance in understanding the Kdrive architecture and patiently explaining Render. Thanks also to Anders Carlsson for help in implementing parts of the KAA and Rage 128 Blend work. Thanks to Keith Packard, Carl Worth, and Deborah Anholt for many rounds of editing of this paper. Finally, many thanks to LinuxFund for sponsoring my work on Kdrive.

## Bibliography

[1] Robert W Scheifler and James Gettys, *X Window System 3d ed.*, Digital Press. 1992.

[2] Keith Packard, *Design and Implementation of the X Rendering Extension*, FREENIX Track, 2001 Usenix Annual Technical Conference. http://keithp.com/~keithp/talks/usenix2001/xrender/, (2001)

[3] Keith Packard, *A New Rendering Model for X*, FREENIX Track, 2000 Usenix Annual Technical Conference. 279-284 (2000)

[4] Keith Packard, *The X Rendering Extension*, The XFree86 Project, Inc.

[5] Mason Woo, Jackie Neider, Tom Davis, & Dave Shriener, *OpenGL Programming Guide 3rd ed.*, (1999)

[6] Mark Vojkovich and Marc Aurele La France, *XAA.HOWTO*, The XFree86 Project, Inc.

# How Xlib is Implemented (And What We're Doing About It)

Jamey Sharp
Computer Science Department
Portland State University
Portland, OR USA 97207–0751
jamey@cs.pdx.edu
http://xcb.freedesktop.org

## Abstract

The X Window System is the *de facto* standard graphical environment for Linux and Unix hosts, and is usable on nearly any class of computer one could find today. Its success is partially due to its flexible, extensible design.

Unfortunately, as research proceeds on cutting-edge window system functionality, the brittleness of the underlying software is a critical impediment to progress. Xlib, the client-side implementation of the network protocol that underlies X, is one source of these issues. Many developers working on new features in the X protocol are discovering that Xlib requires changes to support these features, but Xlib makes those changes difficult. For more than 15 years, new features have been added to Xlib by accretion, rather than with careful design.

We discuss the implementation of Xlib and analyze some specific difficulties in it that cause problems in understanding and maintaining this code base. We also present our current work on migrating the X Window System to a more maintainable, carefully designed architecture.

## 1 Introduction

At the core of the X Window System [SG86] is a network protocol, allowing any number of X applications running on far-flung machines to interact with a single keyboard, mouse, and monitor. Nearly all applications currently use Xlib [SGFR92], a C library dating to the mid-1980s, to interact with the X protocol.

Software developers have collectively learned a lot about the engineering of software in the decades since X was created. This fact explains some, though not all, of the difficulties that users and developers experience when working with X. (Other issues are explained later in this paper.) Our previous work on a new X-protocol C Binding [MS01] and subsequently an Xlib Compatibility Layer [SM02] were efforts to apply current best practices in software engineering to the core of the client-side implementation of the X protocol, with goals of improving the usability of X in several cases:

- Resource-constrained environments, such as PDAs.
- Developers wanting to understand how X works.
- Developers implementing X protocol extensions.
- Users desiring better-performing applications.

Xlib spans more than 400 source files, contains more than 150,000 lines of code, and compiles to a roughly 750kB shared library on Linux/i386. On a typical Linux system, this puts Xlib among the top libraries as measured by code size.

This paper has several major parts: a tour of Xlib, an explanation of several current efforts to improve X, some observations on software engineering, and a brief glimpse at the future.

## 2 The Architecture of Xlib

As we are not aware of a comprehensive tour of the Xlib source, we present one here.

Xlib stores more than a kilobyte of data about each X server connection in a structure named a `Display`. This includes

- The file descriptor and other information about the transport.
- Other file descriptors to monitor for new data.
- An assortment of values cached from the server.
- Pointers used for internal memory management.
- Function pointers to hooks.

The hooks in Xlib allow extensions and applications to modify the way Xlib handles

- Thread synchronization.
- Conversion between wire protocol and C structures.
- XID allocation.
- Management of graphics contexts and server fonts.
- Buffer flushes.
- Connection close.

When an X client successfully establishes a connection to a server, the server sends several hundred bytes

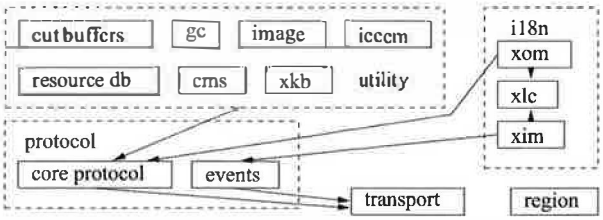



Figure 1: Notional Xlib components: arrows indicate "using" relationships

of information about the capabilities of the display. Xlib copies some of that information into the Display; the rest is kept in other structures pointed to by the Display.

Once connection setup is complete, requests from the client and responses from the server are the fundamental elements of the X protocol. Requests always have a "major opcode" identifying the kind of request, and a length field that measures the number of four-byte words needed to contain the entire packet. Responses come in three forms: replies, events, and errors. Replies and errors are sent in response to requests, while events are sent spontaneously.

Only some request types call for a reply, but for any of those requests, a reply is always sent, unless an error is sent instead. Errors can also occur for other requests. Since X has a network-transparent protocol, it may be run on high-latency connections such as dial-up or DSL Internet links, and on these links replies take long enough to arrive that the delay may have a noticeable effect on application performance. In analyzing this aspect of X performance, we speak of round-trip latency, and look for techniques to avoid or hide it [PG03].

Xlib was originally written without much concern for type-safety, but with great care to minimize the number of function calls. As a result, the C preprocessor gets heavy use when compiling Xlib. Many of the most commonly executed statements in Xlib are macros.

## 2.1 Xlib Layers

As shown in Figure 1, Xlib can be thought of as having several distinct layers and components. An analysis of the size of these notional components is in Section 6.2. Unfortunately, within the code the boundaries are not as clear as is suggested by the figure. In fact, we are not aware of any previous efforts to describe Xlib in this manner. However, these boundaries will be useful in our analysis of Xlib, and correspond to the vocabulary commonly used in conversation among Xlib developers.

### 2.1.1 Transport Layer

The transport layer is responsible for conveying requests and responses between the X client and server. This layer is independent of the semantics of those packets.

Much of the code in this layer comes from xtrans, a module comprising several `.c` and `.h` files. Code that uses xtrans instantiates it in one of several variants by defining an appropriate C preprocessor macro before including not just `Xtrans.h`, but also `transport.c`. Available variants are X11, XSERV, Xim, FS, Font, Ice, TEST, and LBXPROXY: while there are occasional subtle differences, these variants differ mostly in symbol names.

Xlib uses two variants of xtrans, X11Trans and XimTrans, meaning that all of the code in xtrans is linked into Xlib twice. The first variant is used in the transport layer, while the second is used by X input methods. We focus on the first of these here.

Nothing in the extension libraries and applications that we have tested uses X11Trans directly. Within Xlib, calls to X11Trans are confined to four source files: XlibInt.c, OpenDis.c, ConnDis.c, and ClDisplay.c. As a result, rewriting Xlib to eliminate references to X11Trans is a relatively straightforward task.

### 2.1.2 Protocol Layer

On a Cray supercomputer, memory is not addressable in single-byte increments: in fact, it is addressable only in 64-bit increments. In X requests and responses, however, 32-bit values are aligned to 32-bit boundaries, 16-bit values to 16-bit boundaries, and so on. On a Cray, then, accessing individual components of a request or response is inefficient.

Xlib is designed to unpack the wire protocol data from responses into structures that may be efficiently accessed by the host, and to similarly pack data into requests. Many parts of the protocol are represented by a pair of structures in Xlib, namely the wire protocol structure and the host structure. By convention, the names of wire structures are of the form x*ID*Req for requests and x*ID*Reply for replies, with various *ID*s, and in general components of these structures correspond only to arguments to functions. The core event wire type is xEvent and corresponds to host structures named X*ID*Event. The core error wire type is xError, and corresponds to XErrorEvent.

For all of the wire protocol structures, C preprocessor symbols with names like `sz_xEvent` are defined equal to the number of bytes that the structure occupies on the wire. Request structures also have their opcode (major if core, minor if extension) in `#defines`. Structures have fixed-length parts, but no variable-length representation.

When Xlib accesses a Display, perhaps to construct a request or process a response, it must protect the Display against simultaneous accesses by other threads. The LockDisplay and UnlockDisplay macros are provided for this purpose. All access to a Display must be bracketed by a LockDisplay and UnlockDisplay pair.

To deliver a request to the server, Xlib must allocate both a sequence number and a block of memory to construct the request in. Both of these allocations are handled by the GetReq macro or its variants: GetResReq, GetEmptyReq, and GetReqExtra. Any variable-length parts of the request are then delivered with the Data macro, its variants Data16 and Data32, or the _XSend function. The Data macro will copy into the output buffer if there is enough room, or call _XSend with its arguments otherwise. The _XSend function uses the writev system call to write both the buffer and the extra data in one system call, without further copying. Finally, if a reply is expected from the server, _XReply is called to wait for the round-trip to complete so the library can return the data to the caller.

The Display must be locked during this process for two reasons:

1. The block of memory is allocated directly from the output buffer of the Display, and then written to by subsequent instructions.
2. The "current sequence number" is stored in the Display. It is updated by GetReq, but then if a reply is expected, that number must remain constant until _XReply is called.

Xlib offers a feature called synchronous mode. The X manual page says,

> Since Xlib normally buffers requests to the server, errors do not necessarily get reported immediately after they occur. This option turns off the buffering so that the application can be debugged. It should never be used with a working program.

Inside of Xlib, this is implemented with a function pointer (synchandler) stored in the Display structure and called at the end of every protocol stub. (The function pointer is occasionally used for other purposes, and at those times the original pointer is saved in the saved-synchandler pointer.) The SyncHandle macro hides the details of that function call.

In the core X protocol, any drawing request requires a graphics context (GC) identifier as one of its parameters. GCs encapsulate a good deal of state that often should persist between drawing requests, and so are both convenient and bandwidth-efficient. A small challenge lies in creating a convenient C-language interface for setting the many properties of a GC, and Xlib provides a collection of functions allowing the application to set one property at a time. Of course, it would be inefficient to generate a request every time any of these functions is called, so Xlib only constructs a request modifying the GC when some other request that uses that GC is about to be constructed, and then only if the properties of the GC are truly different on the client than on the server. The FlushGC macro hides these details.

It is frequently convenient to have a standardized set of strings for inter-client communications, but preferable to compress the representation of the strings. The X server maintains a list of "atoms", which are numbers that are uniquely mapped to strings for the lifetime of the X server process. Applications may use the InternAtom request to get (or create, if necessary and desired) the atom associated with a string, and the GetAtomName request to get the string for a particular atom. Since most applications participate in a significant number of inter-client communications using a number of atoms, Xlib caches part of the mapping held by the server. The size of the cache is currently hard-coded to be 64 atoms.

### 2.1.3 Utilities Layer

From the point of view of the X protocol, any code not providing transport or stubs for requests and responses is utility code. Some functions contain both protocol and utility functionality. XPutImage is an example of this. In addition to delivering image data to the X server, XPutImage splits the image if it exceeds the maximum request length; byte-swaps individual pixels; and does other conversions as needed.

## 2.2 Xlib Modules and Interfaces

Since Xlib contains a wide variety of distinct interfaces and functionality, here is a list of the major components.

*locking:* Thread synchronization primitives, serving to protect the Display structure.

*transport:* Low-level communication with one or more X servers, encapsulating support for a variety of reliable stream protocols and handling buffering and connection setup.

*core protocol:* Stubs for all requests in the core X protocol.

*cut buffers:* Utility functions for manipulating selections, also known as cut buffers.

*gc:* Graphics context cache, allowing applications to change the many properties of a graphics context while only generating requests when the graphics context is used.

*image:* An assortment of utility functions for operating on client-side image buffers, including loading bitmaps from files.

*events:* Xlib provides a variety of event queue search functions, matching on criteria such as whether an event is related to a specific window.

*icccm:* The Inter-Client Communications Conventions Manual [Ros] standardizes such things as communicating window titles to window managers. Xlib provides utility functions for the operations documented in the manual.

*region:* The region data structure describes arbitrary sets of pixels. It has been re-implemented several times in different parts of the X Window System because it has broad utility; but in Xlib it is not useful for much more than setting arbitrary graphics context clipping regions.

*resource db:* Xt [AS90], the original X Toolkit, gets application preferences from the "resource manager database", which is stored as a property on the root window of the first screen of each server. Xlib provides the low-level support for this mechanism, although it is not generally used in new applications or toolkits.

*xom:* The X Output Methods provide support for displaying text encoded using ISO 2022, also known as ECMA-35. This standard, which predates Unicode, provides escape sequences for switching between character sets.

*cms:* A "color management system" (CMS) provides mechanisms for transforming colors between differing color spaces, including device-dependent spaces. With a CMS, users may calibrate scanners, printers, and monitors so that colors appear identical. Xcms [Ber95] can only use monitor calibration data.

*xkb:* The X Keyboard Extension generalizes the keyboard model of the core protocol. Xlib contains code for interacting with this extension if supported on the server and compatibility code for applications and servers unaware of XKB.

*xlc:* The X Locale implementation maps strings between a variety of encodings and formats. Supported formats include C strings, wide character strings, and UTF-8, among others.

*xim:* The X Input Methods allow a user to input text using alphabets not physically present on his or her keyboard. Japanese text, for instance, can be input by typing the phonetic pronunciation of a word and searching for a character with that pronunciation and the intended meaning.

# 3 Some Xlib Issues

Given skill at reading software written in C and a little understanding of the X protocol, almost any individual Xlib source file may be understood without too much effort. Difficulty in comprehending the whole is due primarily to bulk of Xlib. The broad scope of Xlib, together with the engineering needed to make it work on computers of the 1980s, led to a large implementation, and this implementation has grown dramatically with time. Also, since all of the source of Xlib was written by hand, revising design decisions that affect any significant part of the code is an exceedingly difficult task.

As a brief example, xtrans (covered in Section 2.1.1) is included in Xlib in two forms: X11Trans and XimTrans. These two forms produce nearly identical compiled code, so they are essentially redundant. They are compiled into Xlib using a technique strongly discouraged in C programming: the C preprocessor is used to include source from a .c file into another, nearly empty, .c file. It takes a good deal of time for even an experienced programmer to understand this particular component.

## 3.1 Inflexible Implementation

Xlib provides two supported ways of accessing information stored in the Display structure: functions and macros. This means that offsets in the Display structure must remain constant to maintain binary compatibility with any code using these macros. Unfortunately, the specification [SGN88, P12] says:

> The macros are used for C programming, and their corresponding function equivalents are for other language bindings.

Because of this advice, X applications, which are predominantly written in C and C++, generally use the macro variants of the accessors. They leave Xlib developers little flexibility to revise design decisions. For binary compatibility, neither the macros nor the data structures that they access can ever be changed.

This problem is aggravated by the fact that Xlib has traditionally installed the "private" header file Xlibint.h alongside the public header files intended to be used by applications and libraries. It was made available for the sole purpose of providing extension libraries access to functions, macros, and data structures convenient for processing X protocol messages. However, toolkit and application writers have taken advantage of the direct data structure access, and now some code depends on vagaries of Xlib implementation.

## 3.2 Unpredictable Requests

Immediately after connection setup, Xlib automatically generates several requests regardless of whether the application needs those steps taken. The steps are

- Setting up support for requests larger than 256kB.
- Creating a default graphics context for each screen.
- Retrieving the resource database.
- Initializing the XKEYBOARD extension.

If the server supports both the BIG-REQUESTS and XKEYBOARD extensions, then this process will block application startup for the duration of five round-trips to the server. On modern configurations, the resource database can be easily 30 times as large as the data returned by the server during connection setup.

This is only one example of the larger problem that Xlib generates X protocol requests that are not obviously related to the needs of the application. Another example is that it is impossible to use the public API of Xlib to

send a GetWindowAttributes request without simultaneously sending a GetGeometry request. Between this sort of opportunistic request generation and the assortment of caches implemented in Xlib, an application cannot effectively predict what protocol requests Xlib will produce on its behalf. That, in turn, suggests that these features would have been more useful as an extra layer built on top of functions providing direct control over protocol generation.

## 3.3 Threading

One aspect of Xlib that is not at all straightforward is the support for multi-threaded applications. It is perhaps partly due to the complexity of this aspect that the only application we could find to test Xlib in a multi-threaded setting is ico, a sample program from the reference implementation. Whatever the reasons, the thread support in Xlib is poorly tested (though we have yet to be able to demonstrate any actual faults), almost never used, and very difficult to understand in depth.

This is unfortunate. Threads provide a powerful way to organize computation and I/O with a minimum of programmer effort. Major X applications like Mozilla use threads to great effect, yet are careful to make all Xlib calls from a single thread. Users see this as their applications freezing occasionally, for example while rendering a new web page in a browser.

This section provides a sample of confusing thread synchronization situations in Xlib, with explanations of how those situations work. This is by no means an exhaustive list, however.

### 3.3.1 XLockDisplay

Nearly every function in Xlib invokes the LockDisplay and UnlockDisplay macros. Between LockDisplay and UnlockDisplay, the function is permitted to make any changes to the state of the Display structure. This lock is implemented using a mutex.

In addition, the documented interface to Xlib includes XLockDisplay and XUnlockDisplay. The documentation says that XLockDisplay may be called multiple times from the same thread without deadlock, and XUnlockDisplay must be called an equal number of times before the display is actually unlocked. The UNIX98 standard calls this a recursive mutex; it is also known as a counting mutex.

However, XLockDisplay and XUnlockDisplay are not implemented using a recursive mutex. Instead, they use condition variables together with a non-recursive ("fast") mutex. Pseudo-code for this algorithm is provided in Figure 2.

A fair amount of experience at working with concurrent software is required to understand this algorithm. While it does seem to satisfy its specification, a clearer

```
_XInitDisplayLock
  level = 0

LockDisplay
  lock mutex
  while level > 0 && thread != self
    wait on condition variable

UnlockDisplay
  unlock mutex

XLockDisplay
  LockDisplay
  ++level
  thread = self
  UnlockDisplay

XUnlockDisplay
  LockDisplay
  --level
  if level == 0
    wake up all waiting threads
  UnlockDisplay
```

Figure 2: Xlib thread-synchronization primitives

equivalent is desirable. It would be preferable to use a standard mutex to implement this counting mutex, and in fact we devised and tested an algorithm to do that.

Unfortunately, our algorithm still was not perfect under the metric of clarity. That algorithm had enough problems to fall back to an even simpler plan: just use recursive mutexes. The UNIX98 standard offers them, and using them reduces all four of the synchronization primitives in Xlib to single-line calls to pthread_mutex_lock or pthread_mutex_unlock.

### 3.3.2 GetReq and _XReply

Anyone reading Xlib sources could be forgiven for thinking that the constraints are very strict on functions that may be called in between calls to the GetReq family of macros and to the _XReply function. After all, _XReply discovers the sequence number that it is waiting for by checking the value in `dpy->request`, which was set by GetReq, so clearly nothing should be allowed to touch that value during that interval.

Now note that several functions call Data (Xlibint.h) between GetReq (Xlibint.h) and _XReply (XlibInt.c); Data may call _XSend (XlibInt.c); _XSend may call _XWaitForWritable (XlibInt.c); and _XWaitForWritable may release the display lock while calling select. It should seem disturbing that the display lock might be released during this period where `dpy->request` must not be touched.

```
#include <X11/Xlib.h>
#include <pthread.h>

static char atom_name[32768];

void *event_loop(void *arg)
{
  Display *dpy = arg;
  XEvent evt;
  while(1)
    XNextEvent(dpy, &evt);
}

void main(void)
{
  Display *dpy;
  pthread_t event_thread;
  Atom atom;
  int i = sizeof(atom_name) - 1;
  atom_name[i--] = '\0';
  while(i >= 0)
    atom_name[i--] = '1';
  XInitThreads();
  dpy = XOpenDisplay(0);
  pthread_create(&event_thread, 0,
                 event_loop, dpy);
  atom = XInternAtom(dpy, atom_name,
                     False);
}
```

Figure 3: Does this program work? (Yes.)

This situation is very difficult to reason about: The chain of calls is long, the functions are complex and far apart in the source, and the circumstances under which the display lock might be released are complicated.

Consider Figure 3. In this example we send a request, InternAtom, which will send back a reply; and we ensure that the atom name sent in the request is larger than the output buffer in Xlib (which defaults to 16kB). That will cause _XSend to be called before _XReply in XInternAtom. However, prior to sending this request, we ensure that XNextEvent is waiting in another thread to read any responses that become available on the wire.

What if the event thread reads from the connection while _XSend is trying to deliver the InternAtom request? Could XNextEvent read the reply accidentally? If so, it would not know what to do with it.

The trick is that the reply cannot come back until the entire request has been written, and _XSend was carefully written so that it would return with the lock held and without reading as soon as it finishes writing its single request. Therefore, no other thread has an opportunity to read the reply.

This leaves open the question of whether there is some other fault in this part of Xlib. For instance, perhaps GetReq could be invoked from another thread while _XWaitForWritable has the display unlocked, and perhaps the sequence number stream would be corrupted as a result. We continue to inspect the source of Xlib for cases like this.

## 4 Starting Over: XCB

We wanted a simpler, smaller base for X development than Xlib, so we wrote XCB. As we explained in [MS01],

> XCB is intended to be a simple and direct binding of X protocol transactions to C language function calls, with the minimum amount of machinery necessary to achieve this aim.

As a result, the XCB interface consists of little more than functions that send requests to the X server and a bit of machinery to handle the responses. This makes its interface much smaller than that of Xlib. The most noticeable benefits of this limited interface are that XCB has much less code than Xlib and a much simpler implementation. When built with reasonable compiler optimizations, XCB is 26kB compared with 750kB for Xlib.

Layers and components are organized in XCB in a manner similar to that presented for Xlib in Figure 1. In contrast with Xlib, however, the boundaries are more rigidly enforced, and in fact XCB offers only a minimal utilities layer. Most utility functionality is expected to be provided by separate libraries.

### 4.1 X Protocol Description Language

The C programming language is not ideal for the task of describing the X protocol. Patterns emerge in the code that cannot be eliminated using C language constructs, and logically related definitions are forced to be split between header and source files. These issues make maintenance difficult and cause problems for those attempting to understand the functioning of any particular protocol request.

We created a domain-specific language to describe the essence of the protocol. The immediate benefits are these: all information about a request can be found in one convenient bundle, and the implementation is easy to change without changing hundreds of protocol stubs by hand. Over the longer term, these protocol descriptions have further value because they may be reused in a number of ways, including but not limited to automatically producing

- Bindings for other languages.

- Protocol documentation.
- A text representation for use in debuggers like xscope and xev.
- Server-side protocol bindings.

This re-usability has been important in our work, and we discuss it in more detail in section 5.2.2.

## 4.2 Constructing Requests

In the initial development of XCB, we followed the lead of Xlib on a number of implementation choices. For the most part, these choices were harmless, but one example is instructive. Like Xlib, XCB needs to allocate blocks of memory for request construction. Like the GetReq family of Xlib macros, we initially allocated these blocks directly out of the output buffer. Everyone involved at that time thought this was a perfectly reasonable choice, if for no other reason than that it avoids copying data from some other block into the output buffer.

Eventually, however, we came to feel that this interface was excessively constraining on the implementation of the transport layer. It assumed that memory allocated for requests did not need to be deallocated, and it required that the output buffer be protected against concurrent access for the entire duration of request construction.

We decided instead to allocate request buffers on the stack. This had one immediate advantage: every protocol stub for XCB requires only one function call to deliver a request to the X server, unifying the functionality of GetReq, Data/_XSend, and _XReply. That function is called XCBSendRequest, and it hides the details of computing the length field of requests, including requests larger than 256kB.

In the end, replacing the GetReq model with XCBSendRequest simplifies the interface between protocol stubs and the transport layer of XCB and reduces the duration that locks are held and the number of function calls in many protocol stubs. This change even reduces the size of the compiled code by a small margin. A similar approach would have produced very good results for Xlib as well, at the cost of an extra function call for many protocol stubs, which in the past would have been considered prohibitive. Unfortunately, since Xlib protocol stubs are entirely hand-coded and use macros like GetReq that have all of the problems of inflexible implementation discussed in Section 3.1, changing the design of this part of Xlib at this point would require massive effort.

# 5 Porting Xlib to XCB

Given the many benefits of XCB, we would like to see applications and libraries use XCB as their low-level interface to the X server. However, nearly every X application in existence uses Xlib, whether through one of the many toolkits or directly. Quite a few of these applications, and some of the toolkits, are closed-source; in some cases the source has been lost. As for the open source X applications, there are too many to count, let alone port.

Clearly, some means of transitioning Xlib applications to XCB is needed. It should be fully binary compatible with existing Xlib-based libraries and applications while allowing XCB to manage the connection to the X server. In [SM02], we described such a library, and called it XCL: the "Xlib Compatibility Layer".

XCL was intended to be a drop-in, source compatible replacement for Xlib, adding an extra interface to allow access to the underlying XCB connection. As it turned out, binary compatibility was easy to achieve as well. The intent was that applications would be able to take advantage of some of the benefits of XCB without any modifications, and then use more features of XCB as portions of the source were ported. As a practical matter, this would mean that applications and libraries that use Xlib may be mixed with those that use XCB.

## 5.1 Some XCL Issues

The first attempted implementation strategy for XCL was to start from scratch, adding support for more of the Xlib interface as applications that actually needed it were discovered. This effort was not expected to result in a full re-implementation of the Xlib interface, which is much too large to create with a reasonable amount of effort. In fact, that issue was a show-stopper. Every part of the Xlib interface is probably used by some application, somewhere, and unless an XCB-based version of Xlib supports every one of those applications, the traditional Xlib must continue to be maintained in parallel. The duplicate effort inherent in such a plan led to the eventual dismissal of the XCL implementation strategy.

## 5.2 Some Lessons from Current Work

We are now on our second try, and are approaching the problem from the opposite side. Beginning with the full freedesktop.org implementation of Xlib, we are stripping out and re-writing small parts. This led within a few days to a prototype XCB-based library that supported the full Xlib interface, as well as having limited support for the extra XCL API. The tradeoff is that it is not optimal in code size, clarity, or other measures.

In the version of Xlib at freedesktop.org, the directory layout of the source is notably different than the traditional style. Developers at freedesktop.org have converted the build system from Imake to autoconf, automake, and libtool, and at the same time adopted a layout familiar to anyone working with modern open source software. Public headers may be found in `include/`,

while internal headers and all source code are in `src/`. In addition, while all core X software has traditionally been maintained together in a single source tree larger than 600MB, at freedesktop.org it has been split into many smaller modules and several CVS repositories. Xlib is in the `X11` module in the `xlibs` repository.

### 5.2.1 Transport Layer

Our current work on Xlib began with a focus on the transport layer.

Because of the mismatch between the internal architectures of Xlib and XCB, we provide a minimal set of hooks in XCB so that Xlib may make use of the transport layer of XCB in place of xtrans. (This interface remains a work in progress; we hope to find a clean design that could be useful to callers besides Xlib, but such a design is not yet apparent.) These hooks enable XCB to replace xtrans in a manner transparent to everything other than four Xlib transport-layer source files.

### 5.2.2 Protocol Layer

Replacing the transport layer with XCB provides notable improvements in code size and clarity, as we report in section 6.2. Yet replacing the protocol layer with XCB offers much more significant gains, at the cost of quite a bit of additional development effort. Fortunately, a significant part of this extra effort is already done: The old XCL work amounted to little more than glue code between the core protocol interfaces of Xlib and XCB.

Many functions in the protocol layer serve exactly the same purpose as their counterparts in XCB. In XCL, we built part of the infrastructure needed to automatically generate the code for these functions. That infrastructure combined information from several sources to produce its stubs:

- The descriptions of the protocol from XCB.
- A machine-readable description of the Xlib interface.
- A little bit of hard-coded knowledge about how to convert between data types used by XCB and Xlib.

This illustrates one benefit of encoding knowledge, like the structure of the X protocol, in a domain-specific language. If done well, that knowledge is reusable in a variety of projects.

Other functions require careful inspection when porting them to XCB, and are not expected to benefit from automatic code generation.

## 6 Results

We have tested our version of Xlib on real workloads, including:

- Mozilla, with the Xt-based plug-in Adobe Acrobat Reader.
- Many standard KDE and Gnome applications.
- A variety of window and display managers.

We also ran part of the X Test Suite, which is a comprehensive test suite of Xlib and the X server, created from the well-documented specifications for both.

The three metrics of interest in comparing traditional Xlib with an XCB-based Xlib are

- Correctness: has the behavior of the code changed?
- Code size: do any changes in code size justify the effort required to achieve them?
- Performance: what effect has the work had on speed?

## 6.1 Correctness

For our real workloads, we initially found quite a few bugs both in our version of Xlib and in XCB. However, a rapid series of small fixes resulted in the ability to run a complete desktop environment using XCB. This software is still in the debugging phase, but that phase is mostly done and proceeding well.

### 6.1.1 Observed Causes of Bugs

To transform Xlib into a library built around XCB, the semantics of Xlib must first be well understood, so that those semantics may be maintained. For reasons typified by the examples in Section 3, that task alone is nontrivial. Most bugs in the new version of Xlib result from failures to understand the intended semantics of functions that we have replaced. Naturally, the remaining bugs are due to a failure to correctly re-implement the original semantics.

### 6.1.2 X Test Suite

The X Test Suite reported that there were some defects in error handling. Unfortunately, when tests that failed were re-run individually, they succeeded. It remains unclear where the bug lies, but failure to use the test suite in the manner for which it was designed seems like the most probable suspect.

## 6.2 Code Size

For each source file, the number of lines of code (LOC) were measured by running the file through the C preprocessor, eliminating blank lines and lines from .h files, and counting the remaining lines. This approach was taken because preprocessor conditionals have a significant effect on the number of lines compiled into Xlib, and because it accounts nicely for the inclusion of xtrans. The number of bytes of compiled code due to each source file was computed by examining object files intended for a statically linked library, which do not have the extra code generated for position-independent code (PIC). (The overhead of PIC was deemed uninterest-

| | Xlib | | | | | | XCB | |
|---|---|---|---|---|---|---|---|---|
| Component | LOC | % | Δ LOC | bytes | % | Δ bytes | LOC | bytes |
| locking | 125 | 0.20 | -166 | 1109 | 0.14 | -1348 | n/a | |
| transport | 2100 | 3.35 | -2261 | 17886 | 2.24 | -21264 | 1005 | 11250 |
| core protocol | 6003 | 9.57 | -560 | 58839 | 7.36 | -3683 | 2619 | 19665 |
| extensions | no change | | | | | | 1442 | 15080 |
| Total | 62702 | 100.00 | -2987 | 799678 | 100.00 | -26295 | 5066 | 45995 |

Table 2: Code size for Xlib with XCB

| Component | LOC | % | bytes | % |
|---|---|---|---|---|
| locking | 291 | 0.44 | 2457 | 0.30 |
| transport | 4361 | 6.64 | 39150 | 4.74 |
| core protocol | 6563 | 9.99 | 62522 | 7.57 |
| cut buffers | 99 | 0.15 | 599 | 0.07 |
| gc | 370 | 0.56 | 2494 | 0.30 |
| image | 1027 | 1.56 | 9550 | 1.16 |
| events | 1112 | 1.69 | 6762 | 0.82 |
| icccm | 1160 | 1.77 | 8847 | 1.07 |
| region | 1290 | 1.96 | 9504 | 1.15 |
| resource db | 2554 | 3.89 | 65719 | 7.96 |
| xom | 2891 | 4.40 | 24938 | 3.02 |
| cms | 6478 | 9.86 | 62188 | 7.53 |
| xkb | 10824 | 16.48 | 104430 | 12.64 |
| xlc | 12323 | 18.76 | 312602 | 37.85 |
| xim | 13615 | 20.73 | 106578 | 12.90 |
| Other | 731 | 1.11 | 7633 | 0.92 |
| Total | 65689 | 100.00 | 825973 | 100.00 |

Table 1: Code size for traditional Xlib

ing for this analysis.) These object files were built for Linux/x86 without optimization. The Unix `size` command was run on each of these object files, and the value from the "dec" column was taken, which includes code, string literals, and any other data. Finally, each source/object file pair was assigned to a component to produce summary results per component.

In Table 1, the number of lines of code and compiled bytes are given for traditional Xlib in two forms: raw, and as a percentage of the total for that library. The revised version of Xlib, together with XCB, is covered in Table 2. In that table, the number of lines of code and compiled bytes are given in raw and percentage form as before, plus the change (Δ) relative to traditional Xlib. Additionally, lines of code and compiled bytes are given for XCB in raw form. Xlib components unaffected by this work were omitted from Table 2.

XCB provides a substantial improvement in code size to the transport layer of Xlib, and in the future is expected to do the same for the protocol layer and extension implementations that have been built on top of Xlib. Yet these improvements comes at relatively little cost in human programmer time: much of the code is automatically generated from straightforward declarative descriptions. Our current work has added less than 700 lines of new hand-written code to Xlib, while making thousands of existing lines irrelevant.

The core protocol layer of XCB was automatically generated from 1,700 lines of protocol description. The current 55 automatically generated Xlib stubs that delegate to XCB average 7 lines each, for a total of 442 lines. Generation of more stubs is planned.

Similar benefits await client-side extension implementations. Current extension implementations span the protocol and utilities layer for the same reasons that the core protocol implementation does. As a result, some portions of these extensions will be good candidates for automatic code generation by the same techniques that we have used in Xlib, and other portions will gain smaller benefits through hand-porting.

XCB itself probably also has opportunities for reduction in code size. We can experiment with different implementations easily, because nearly 80% of the code in XCB is generated automatically. By happy accident, the introduction of XCBSendRequest – made to improve modularity and code clarity – also reduced code size by a small yet noticeable margin. However, the XCB code base is so small already that we have put in only cursory effort to find further savings there.

## 6.3 Performance

User-visible performance with Xlib is expected to be largely unchanged by the transition to XCB. Rewriting Xlib to use XCB has little effect on those portions of Xlib intended to improve performance, such as the various caches. Patterns of communication between the client and the server should be identical in most cases to traditional Xlib. Xlib cannot use the latency hiding features of XCB to re-order requests and still remain within the constraints of the documented Xlib interface. Some slight performance improvements might be anticipated due to better cache utilization and reduced lock contention, but this is not expected to be significant. Informal testing supports the hypothesis that the difference between traditional Xlib and XCB-based Xlib is not apparent to end-users.

## 7 Related Work

At freedesktop.org, others are currently doing experimental work toward redesigning the utility layer of Xlib. This layer is a prime target for code size improvements, because it

- Is more than 80% of Xlib.
- Is more clearly separable into components.
- Overlaps most with common toolkit functionality.
- Contains the least frequently used code in Xlib.

Current efforts focus on the xim, xom, xlc, cms, and xkb components. Together, xim/xom/xlc make up about 44% of the lines of source in Xlib, and about 54% of the compiled size. The cms component contributes roughly 10%, and xkb contributes 15%, to the size of Xlib. The present build system allows Xlib to be built without each of these components, producing a build of Xlib that is about 75% smaller.

According to Jim Gettys [Get03], color management was broken in the XFree86 [xfr] implementation of Xlib for about half a year, and nobody noticed. Apparently that code goes unused.

Unfortunately, some applications and libraries that are in common use do depend on some of this functionality. For instance, Gdk 2.0 uses the xlc component to set window titles. For this reason, entirely removing that code from Xlib is not currently feasible. Fortunately, the character set translation tables that occupy a significant portion of xlc are no longer necessary, as more general libraries such as GNU libiconv provide the same services. One project presently awaiting developers is to remove these tables from Xlib and replace them with whatever implementation of iconv is available.

## 8 Software Engineering Observations

A number of observations may be made about software engineering in general, illustrated by the examples of Xlib and XCB. These observations have been made often in code style guides and software engineering publications, yet code continues to be written that exhibits these problems.

Remember that software is written for two audiences. While a computer must be able to execute the software, it is also necessary that humans be able to read, understand, and modify that software. Much of software engineering comes down to dividing software into manageable chunks, pieces that a human can keep entirely in his or her head long enough to work with them.

Functions should be kept simple, possibly by dividing complicated tasks into several simple functions. Functions that interact strongly should be kept close together, preferably in a single source file. Interactions can and should be weakened where possible through careful modular design, giving callers few opportunities to make mistakes. All of these principles are violated, for example, by the design of GetReq, Data, and _XReply, as explained in Section 3.3.2.

When multiple functions have similar code, a new parameterized function should be created that is the union of the similar blocks. For example, Xlib protocol stubs always call _XSend/Data, _XReply, both, or neither after GetReq. A single parameterized function, similar to XCBSendRequest, would have been better; some reasons were given in Section 4.2.

Functionality like the macros in the C preprocessor should be avoided in modern code. Some reasons for this were given in Section 3.1. Even automatic generation of code, a technique with significant benefits including those described in Sections 4.1 and 5.2.2, has hazards of this sort and should be used with the caution that the generated code should not have redundant similar blocks if the underlying language provides a reasonable mechanism for abstracting them.

Any time a significant chunk of software performs an independent task, that chunk should be an independent module, perhaps encapsulated in its own library. As libraries like XCB and libiconv demonstrate, Xlib contains many components that would have value as stand-alone libraries. Each module and library in a system should be focused on a single reasonably-sized task; have a minimal, orthogonal, and well-defined interface; and be implemented in a readable and maintainable manner.

## 9 The Future of Xlib

Given the benefits of XCB, new X toolkits and applications are anticipated that will use pure XCB rather than Xlib. Legacy Xlib code is expected to slowly migrate to mixed Xlib/XCB and eventually pure XCB. Development of Xlib is expected to slow. Even though the work described in this paper is not ready for widespread release as of this writing, there are already signs that developers are, indeed, moving their focus to XCB.

Xlib development will certainly not cease for some time yet, and is expected to focus on reducing the installation footprint of Xlib. A small number of new features continue to be planned, however. Since the reference implementation of Xlib is open source, Xlib is sure to be supported and maintained until it has no more users.

## 10 Conclusion

We have identified some significant areas of inefficient and confusing design and implementation in Xlib, and presented our efforts to repair this core element of the X Window System. Combined with the efforts of others, we believe that the installation footprint of Xlib may be reduced significantly, while the clarity, maintainabil-

ity, and extensibility of the X client library stack are improved tremendously.

Guiding our efforts are current best practices from the software engineering and formal methods communities, and our work may be taken as a case study in the practical value of these techniques. For that matter, this work would be completely infeasible if the reference X Window System implementation were not open source, and illustrates one of the many benefits of an open development model.

This is an ongoing process. The X Window System has shown an unlimited capacity for extension and innovation. The general techniques of careful modular design, domain specific languages, and others are broadly applicable. We hope we have provoked interest in software engineering in general, and development of the X Window System in particular.

## Availability

XCB and the version of Xlib described here are both hosted by freedesktop.org. XCB is available from `http://xcb.freedesktop.org`. Xlib source is at `http://freedesktop.org/Software/X11`, and can be compiled to use XCB by specifying the `--with-xcb` configure option.

## Acknowledgements

This paper was shepherded by Carl Worth, and his patience and insight have been greatly appreciated.

Many thanks to Keith Packard for helping us comprehend Xlib and the X Window System. We would not have gotten this far without him.

Contributions by Professor Bart Massey have been invaluable as well, providing much-needed guidance and insight for the design and implementation of XCB, as well as mentoring while writing reports on our progress.

Thanks also go to Jim Gettys for his continued support of our efforts.

Finally, Sheridan Mahoney and Mick Thomure provided valuable feedback on drafts of this paper.

## References

[AS90] Paul J. Asente and Ralph R. Swick. *X Window System Toolkit: The Complete Programmer's Guide and Specification*. Digital Press, Bedford, MA, 1990.

[Ber95] David T. Berry. Integrating a color management system with a Unix and X11 environment. *The X Resource*, 13(1):179–180, January 1995.

[Get03] Jim Gettys. Size of Xlib..., October 2003. Web Document. URL `http://pdx.freedesktop.org/pipermail/xlibs/2003-October/000001.html` accessed April 8, 2004 04:30 UTC.

[MS01] Bart Massey and Jamey Sharp. XCB: An X protocol C binding. In *Proceedings of the 2001 XFree86 Technical Conference*, Oakland, CA, November 2001. USENIX.

[PG03] Keith Packard and Jim Gettys. X Window System network performance. In *FREENIX Track, 2003 Usenix Annual Technical Conference*, San Antonio, TX, June 2003. USENIX.

[Ros] David Rosenthal. Inter-Client Communication Conventions Manual. In [SGFR92].

[SG86] Robert W. Scheifler and Jim Gettys. The X Window System. *ACM Transactions on Graphics*, 5(2):79–109, April 1986.

[SGFR92] Robert W. Scheifler, James Gettys, Jim Flowers, and David Rosenthal. *X Window System: The Complete Reference to Xlib, X Protocol, ICCCM, and XLFD*. Digital Press, third edition, 1992.

[SGN88] Robert W. Scheifler, James Gettys, and Ron Newman. *X Window System: C Library and Protocol Reference*. Digital Press, 1988.

[SM02] Jamey Sharp and Bart Massey. XCL: An Xlib compatibility layer for XCB. In *FREENIX Track, 2002 Usenix Annual Technical Conference*, Monterey, CA, June 2002. USENIX.

[xfr] The XFree86 project. Web document. URL `http://www.xfree86.org` accessed April 8, 2004 04:30 UTC.

# Design and Implementation of Netdude, a Framework for Packet Trace Manipulation

Christian Kreibich
*University of Cambridge Computer Laboratory*
*15 JJ Thomson Avenue, Cambridge CB3 0FD, UK*
christian.kreibich@cl.cam.ac.uk

## Abstract

We present the design and implementation of a framework for inspection, visualization, and modification of `tcpdump` packet trace files. The system is modularized into components for distinct application purposes, readily extensible, accessible through programmatic and graphical interfaces, and capable of handling trace files of arbitrary size and content. We include experiences of using the system in several real-world scenarios.

## 1 Introduction

In today's computer networks traffic varies greatly in content and volume, making network analysis a difficult process. Researchers, developers, and system administrators use traffic capturing tools (*sniffers*) to obtain traces of network traffic to gain better understanding of traffic characteristics. Storing traffic flows in a standardized form allows them to investigate the effects of network misconfigurations and programming errors, to perform forensic investigation, to process traffic using appropriate tool chains, and most importantly, to make the occurrence of observed phenomena *reproducible*.

Among the plethora of tools available for this purpose, three freely available ones constitute the defacto standard: the `libpcap` library[1] provides a low-level application programming interface (API) to filter and intercept packets, `tcpdump` presents these packets in textual format, and `ethereal`[2] provides a graphical user interface (GUI) for capturing, filtering, and inspecting packets, supporting a large number of networking protocols and sniffers.

Interestingly, tools that also allow the user to *edit* captured traffic have so far been limited to problem-specific solutions, where the state of the art is disappointing: developers create repositories of frequently unreleased, purpose-specific, throw-away programs, inconveniently written at the `libpcap` level. Yet many of these tools would be useful to a larger audience. Publicly available tools typically have varying calling conventions, and while they normally are fairly easy to use in scripts, they are often not reusable at the API level because their functionality is only available in a standalone executable. These practices violate multiple well-accepted software engineering principles, such as component reuse and the avoidance of cut-and-paste practices, code redundancy, and duplication of effort.

To improve this situation, we present *Netdude*, the *net*work *du*mp data *d*isplayer and *e*ditor, a framework designed to support different packet manipulation paradigms (from APIs to convenient GUIs), emphasizing code reuse, extensibility, and scalability. We think of Netdude as a *workbench* for the creation of new tools, integrating the efforts of other developers. All components presented in this paper are fully implemented and publicly available. We present the architecture of the framework, including design goals and implementation aspects, in Section 2. Section 3 gives usage examples at the API and GUI levels and demonstrates the extensibility of the framework. We describe our experiences in using the framework in a number of real-world scenarios in Section 4. Section 5 discusses the system and presents future work, before Section 6 summarizes the paper.

## 2 Architecture

We first state our design goals for the system. We then present the architecture of our framework, and walk through its components with a detailed explanation of the implementation.

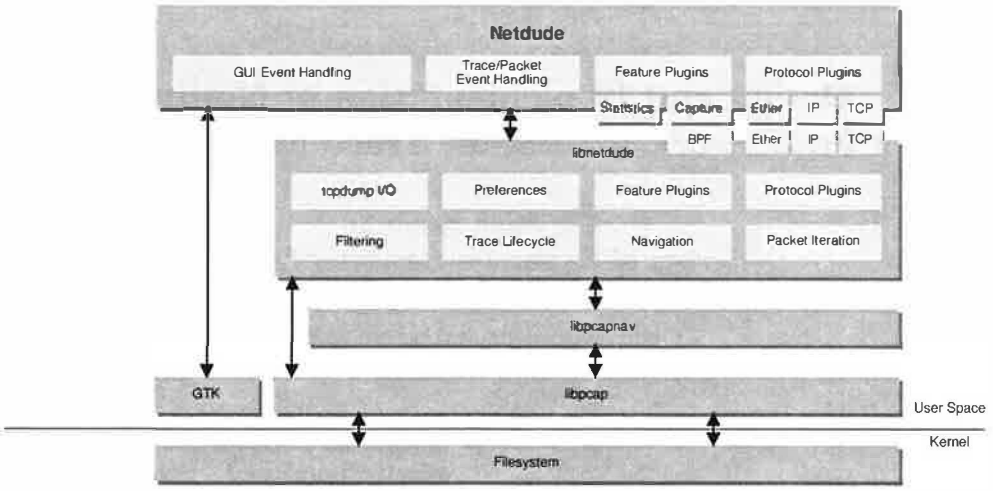



Figure 1: Architecture of the Netdude Framework.

## 2.1 Design Goals

1. MULTIPLE USAGE PARADIGMS
   The user must be able to manipulate trace files at the desired level of interactivity and abstraction. We neither want to enforce only an API, thus asking all users of the framework to become developers, nor a GUI, forcing developers to use a graphical interface that may not be flexible enough. Programmers must find the framework usable at a convenient level of abstraction that allows them to focus on relevant aspects of their algorithms without getting distracted by details of packet reading & writing, trace file navigation, etc. The framework must eliminate the need to hand-write trace file access, filtering, iteration, and protocol demultiplexing code anew for every application.

2. OPENNESS & EXTENSIBILITY
   Our goal is to provide programmers maximum flexibility in making their code interact with our framework. Since networking code is typically written in low-level languages, the programming language must not limit the usability of the framework to a certain language or execution environment. Both programmers and GUI users must have a means to extend the framework using components that they develop themselves or obtain from other developers.

3. SMALL-SCALE EDITING
   The framework must allow the manipulation of packets at a fine-grained level of detail, down to individual bits in the protocol headers and byte sequences in packet payloads. It also must provide the user with means to delete, move, swap, duplicate, and erase packets, and to allow easy saving of changes made to a trace file.

4. LARGE-SCALE EDITING
   The framework must allow the manipulation of arbitrarily large trace files (subject to the maximum allowable file size on the operating system used), particularly files that are much larger than the system memory capacity. Traffic trace files easily reach sizes in the gigabyte range, thus simply loading files into memory at startup is not an option.

The first goal excludes library-only or application-only designs since either would exclude one of the desired user groups. The second goal demands a widely used system programming language; we have decided to implement all library components in the C language to facilitate easy binding to other languages and to provide the largest-possible common denominator. The remaining two goals suggest concentrating the packet manipulation code in a library that can then be used by other programs.

## 2.2 Implementation

These goals lead to a layered architecture, illustrated in Figure 1. In the bottom layer, `libpcap` handles elementary trace file operations: opening and saving traces, sequential reading and writing of packets. The remainder of this section describes the higher layers of the architecture.

### 2.2.1 libpcapnav: Random Packet Access

libpcapnav is a thin wrapper around libpcap that removes the limitations of sequential read access to packets stored in a trace file. Between packet reads, users can jump to arbitrary locations in the trace file, identified by packet timestamps or fractional offsets in the file (e.g., 0.5 identifies the middle of the file). After jumping to a random byte offset in a trace file, the challenge is to properly realign the packet extraction process to the packet sequence contained in the file. The libpcap file format currently provides no markers to identify the beginning of a packet in the file. Even if such a marker was used, it could always occur inside packet data as well. Therefore, a heuristic approach is called for. The algorithm used by libpcapnav is based on the one introduced by the tcpslice[3] tool. Our algorithm uses similar sanity checks on a number of libpcap packet header fields to identify possibly valid *packet chains*, but does not trust a chain of packets to be valid as easily. In cases like trace files containing a file transfer of another trace file (over NFS, via FTP, etc) the danger is to end up in a small chain of libpcap packet headers that actually comprise only the payload transported over the network. To avoid this, libpcapnav scans a window whose size is calculated from the maximum packet size captured in the trace and the maximum length of a chain of packets that the algorithm follows. While scanning, the algorithm keeps track of the chain lengths encountered, keeping only the longest chain.

### 2.2.2 libnetdude: Packet Manipulation

libnetdude is the core of the framework where most of the packet editing functionality is implemented. It provides abstract data types and APIs for handling trace files, regions of trace files, packets, filters, packet iterators, and a few other features described below. libnetdude can handle trace files of up the maximum file size permitted by the file system: it never loads more than a configurable maximum number of packets into memory at any time. Just mmaping regions of the trace file into memory is not an option, since our design goals include the ability to perform arbitrary packet insertions and deletions, and no data can be inserted in the middle of a mmaped memory region. Rather, trace files are edited at the granularity of *trace areas*, whose borders are defined using timestamps or fractional

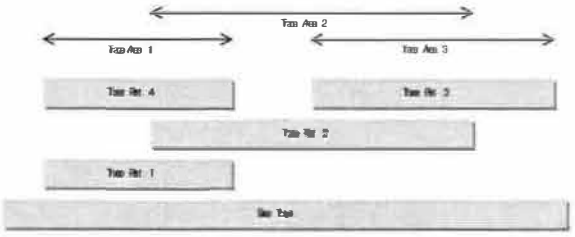


Figure 2: Editing different trace areas causes resulting trace parts to be layered on top of the original trace file.

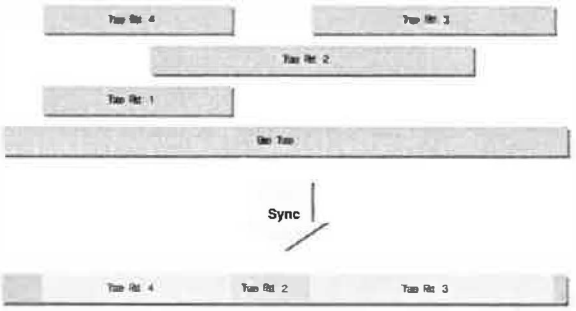



Figure 3: When saving a trace file, the layered parts are flattened onto the original trace file.

offsets understood by libpcapnav. The modified trace areas are stored in temporary storage as *trace parts*, and are logically layered on top of the original trace file, with new trace parts sitting on top of all the other trace parts in the trace area covered. Trace areas and trace parts are carefully maintained by libnetdude, always providing a consistent view of the trace file to the user. Figure 2 illustrates these concepts.

When accessing a packet, the library always uses the trace part in the uppermost layer at the current offset. When a trace file is saved, the trace area layers are flattened onto the original trace file, honoring any inserted or removed packets. The result is a new trace file that contains all modifications made to the original input file. The process is illustrated in Figure 3. The flattening process is performed implicitly through packet iteration: starting at the beginning of the base trace, iteration moves up to areas at higher layers as soon as they are encountered, and returns to lower layers at the end of each trace part.

Note that packet insertion and deletion are straightforward in this approach: the actual composition of packets in a trace area can change but trace parts are still merged onto lower parts at the original

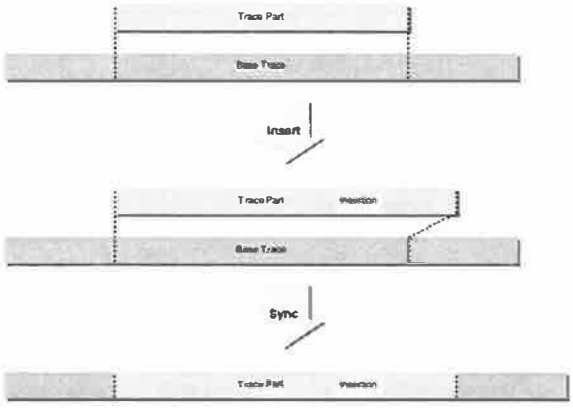



Figure 4: Packet insertion: a trace part is growing in size. When merged onto the base, the original boundaries of the modified trace part are maintained.

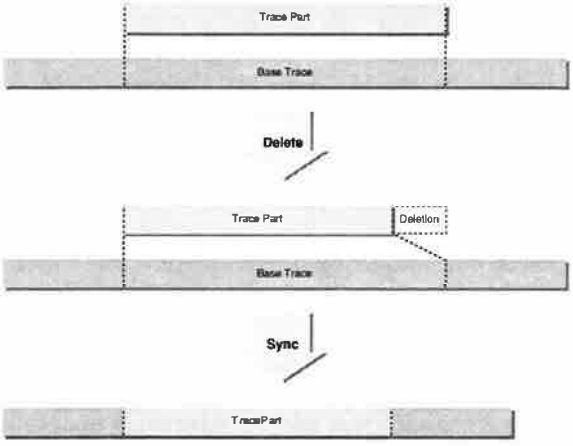



Figure 5: Packet deletion: a trace part is shrinking. Again, the original boundaries of the modified trace part are maintained when merging.

boundaries, regardless of the new higher part's size. This is illustrated in Figures 4 and 5. Furthermore, this approach makes it easy to provide "undo" functionality: removal of the most recent trace part from the stack reverts the most recent modification of the trace file.

In addition, `libnetdude` provides a number of other features:

- A plugin architecture that enables extensibility in two ways: Firstly, *protocol plugins* allow interpretation of arbitrary protocol data. Each protocol plugin provides the knowledge necessary to parse a protocol's header and to select the next protocol in the chain of protocol headers contained in a packet. Secondly, *feature plugins* provide reusable building blocks for functionality that the framework itself does not contain. By linking to other libraries, feature plugins can leverage any functionality accessible at the API level. `libnetdude` provides mechanisms for handling plugin dependencies by allowing plugins to check whether other required plugins are installed. Plugins are dynamically loaded and registered when `libnetdude` bootstraps, using `dlopen` and `dlsym` calls. This happens transparently: plugin authors only need to define a number of well-known functions to make their plugin's capabilities known.

- Structured packet content: once analyzed, packet contents are represented as a sequence of protocol headers. When a packet is initialized, `libnetdude` starts the packet analysis process by consulting the data link type given in the `libpcap` header of the trace file. The corresponding protocol plugin is queried, and control of the analysis is passed to that plugin and then onward as this plugin sees fit. The data structures representing the packet data are created on the fly. Analysis stops when no plugin for a given protocol can be found, or when a plugin does not need to pass analysis on to another protocol. Once the process is finished it is easy to obtain, say, the TCP header of a packet. Nested protocols (such as IP in IP) and arbitrary tunneling are supported. Developers thus need no longer write their own protocol demultiplexers for each application.

- Access to the familiar `tcpdump` output: `libnetdude` can associate each open trace file with its own `tcpdump` process through a bidirectional pipe. The user can then obtain `tcpdump` output at the granularity of individual packets with a single function call[4]. Full control over the output format is preserved by allowing configuration of `tcpdump`'s command line options. Since `libnetdude` can be configured to use any locally installed `tcpdump` executable, changes made to `tcpdump` remain visible inside the framework.

- An observer/observee API for objects like trace files, packets, packet iterators, packet filters, and trace parts. This feature allows seamless integration of the library into the surrounding application, without exposing unnecessary internal state. Users can register callbacks that are invoked when certain events occur in

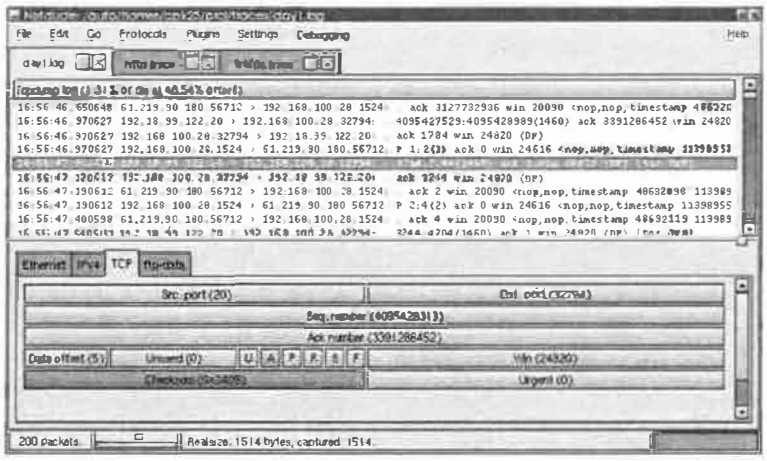



Figure 6: Main window of the Netdude GUI, with three trace files opened, 200 packets of the first file loaded into memory, and the TCP header of the selected packet displayed. The red highlight indicates that the TCP checksum in this packet is incorrect.

the monitored items, such as packet insertions and deletions, trace navigation, or advances in packet iteration.

### 2.2.3 Netdude: GUI Frontend

The Netdude framework provides a GUI application that leverages the functionality `libnetdude` provides. The main window is shown in Figure 6. The application provides graphical interfaces for all underlying abstractions: users can open and save trace files, navigate to arbitrary locations in the trace, inspect packets, configure trace areas to which packet modifications are applied, modify protocol header fields and payload content, and access add-on features through installed feature plugins.

The plugin concept of `libnetdude` is mirrored at the GUI level: protocol plugins allow the visual presentation of a particular protocol's header data, while feature plugins provide GUI access to underlying functionality. The visual representation of header data is entirely up to the plugin author: fixed-width cells can be rendered in a tabular layout, header fields can be color-coded depending on the field value, whereas string-based data may be better represented using list or tree elements.

In order to ensure good performance of trace file operations regardless of the file size, the application relies on `libnetdude`'s approach of limiting the number of packets loaded into memory to a configurable number. When the user jumps to a different location in the trace file, up to this number of packets are loaded into memory and presented as a list in the GUI. Individual packets are analyzed by selecting them from that list. The user can then browse the protocol data in a notebook containing one tab per protocol header contained in a packet. When no plugin can be found to visualize a protocol header, a fallback hex editor allows for inspection and modification of packet data using two different modes: a hexadecimal mode that presents each byte in ASCII and hex, and a pure ASCII representation suitable for text-based data.

## 3 Framework Usage

We illustrate the usage of the framework at the GUI and API levels using two examples: iterating over packets, and accessing selected protocol headers in a packet. Figure 7 shows `libnetdude` code for these scenarios. To illustrate the flexibility of the plugin mechanism, we then present a few feature plugins for `libnetdude`.

### 3.1 Packet Iteration

Using `libnetdude`, packet iteration is done in two steps: first the area of the trace that the user wants to iterate is specified. Then, a packet iterator instance is used in a `for`-loop. In each iteration, the current packet can be obtained from the packet iterator. `libnetdude` differentiates between read-only

```
#include <libnd.h>
#include <netinet/in.h>
#include <netinet/tcp.h>

void iterate_tcp_dports(const char *tracefile)
{
  LND_Trace          *trace;
  LND_PacketIterator  pit;
  LND_TraceArea       area;
  LND_Protocol       *tcp;
  struct tcphdr      *tcphdr;

  /* Obtain a handle to the TCP protocol */
  if (! (tcp = libnd_proto_registry_find(LND_PROTO_LAYER_TRANS, IPPROTO_TCP))) {
    /* Protocol not found -- handle accordingly. */
  }

  /* Open the trace file: */
  if (! (trace = libnd_trace_new(tracefile))) {
    /* Didn't work -- appropriate error handling. */
  }

  /* Set the trace's active area to the second half of the file. */
  libnd_trace_area_init_space(&area, 0.5, 1.0);
  libnd_trace_set_area(trace, &area);

  /* Iterate over all packets in that trace area */
  for (libnd_pit_init(&pit, trace); libnd_pit_get(&pit); libnd_pit_next(&pit)) {

    /* Request the TCP header of the current packet. */
    tcphdr = (struct tcphdr *) libnd_packet_get_data(libnd_pit_get(&pit), tcp, 0);

    /* If a TCP header was found, print its destination port. */
    if (tcphdr)
      printf("Dest. port: %u\n", ntohs(tcphdr->th_dport));
  }
}
```

Figure 7: A `libnetdude` example, iterating the second half of a trace and printing out the destination ports of all TCP packets in that area.

and read/write iteration because packet modifications require the creation of a new trace part for the trace area iterated. In this case, the user can selectively drop packets during the iteration.

Using the GUI, the user first defines the trace area using a dialog. The iteration is then performed implicitly when the user modifies a packet (for example by setting a header field to a certain value, or fixing checksums): the same modification is applied to all packets in the configured trace area, subject to configured packet filters.

## 3.2 Accessing Protocol Data

Using `libnetdude`, the user obtains a handle for the desired protocol by specifying the protocol's layer in the network stack and the identifier of the protocol commonly used at that layer (e.g., `IPPROTO_xxx` values at the network layer). The user then requests a pointer to this protocol's header data in a packet, for the desired nesting level. Note that this querying mechanism does not imply that the protocol must be used at the specified protocol layer in the packet data. The layer is only used to obtain a handle to the data structure representing the protocol in `libnetdude`.

Using the GUI, the user first selects a packet from the list of packets currently loaded into memory. The GUI then provides access to the individual protocol headers contained in that packet. The user selects the desired protocol header and directly manipulates the header's bit fields as visualized by the responsible plugin (e.g., using pull-down menus for fixed-range values, or entry fields for variable fields).

## 3.3 `libnetdude` Feature Plugins

When making new features available at the API and GUI levels, we prefer to first provide functionality in `libnetdude` feature plugins and then add Netdude GUI frontends later on. `libnetdude`'s plugin-based architecture has important implications for developers: consider a programmer who wants to develop an application that uses functionality provided by a feature plugin. When using C, he or she would typically use the plugin's API by including the plugin's header file(s). However, this causes problems at link time due to the late-binding design of `libnetdude`'s plugins: when linking the new application's code to `libnetdude`, the symbol definitions of the required plugins are not available since the object files are only linked in after the library's bootstrapping process completes. The result is an undefined symbol error. Adding the plugin's shared object files at link time is insufficient in case the plugin itself requires other plugins; this approach would quickly result in *all* plugin object files being added at link time, violating our design goal of flexible extensibility.

As a more scalable solution, we ask plugin developers to provide the new functionality in `libnetdude` plugins themselves. This way, symbols are only resolved at runtime (since the plugins are built as shared objects) and linking can remain constrained to the new plugin. Undefined symbols can still be caught at compile time using suitable compiler options, such as `-Werror-implicit-function-declaration` when using GCC.

To make the plugin's functionality accessible, developers have two options: the first is to provide their own executable that initializes the library, queries the plugin, and runs it with appropriate parameters. This only needs a few lines of code. The second option is to use the `lndtool` command line frontend that is provided by `libnetdude`. This tool can be used to query several parameters of the local `libnetdude` installation. As an example, Figure 8 shows how `lndtool` lists installed plugins.

```
cpk25@ghouls:/auto/homes/cpk25 > lndtool --plugins
libnetdude protocol plugins:
-------------------------------------------------
Ethernet                        0.5
ICMP                            0.5
IPv4                            0.5
SLL                             0.5
LLC/SNAP                        0.5
TCP                             0.5
UDP                             0.5
ARP                             0.5
FDDI                            0.5

libnetdude feature plugins:
-------------------------------------------------
BPF-Filter                      0.5
Checksum-Fix                    0.5
PHDL                            0.1
TCP-Filter                      0.1
TCP-State-Tracker               0.2
Trace-Set                       0.1
Traffic-Analyzer                0.3
```

Figure 8: Running `lndtool` to obtain a list of installed plugins.

`lndtool` also provides a command line interface for accessing the plugins, passing command line arguments through to the selected plugin. Examples of `lndtool`'s usage are given in the following selection from feature plugins that have been developed so far:

- Trace Sets: The need to operate on a set of traces occurs frequently. To allow easy reuse of this functionality, we have developed a plugin that manages the life cycle of sets of trace files and provides a mechanism to iterate over trace files contained in such a set.
- TCP Connection Tracking: Most TCP-based packet manipulation requires TCP connection state tracking. This plugin provides such functionality and allows the user to maintain and query connection state for a number of flows, and check whether the three-way handshake or connection teardown were fully observed. The plugin is useful in itself: when run as `lndtool -r tcp-state-tracker <trace>`, it prints the familiar `tcpdump` output but augments each TCP packet's line by including the current connection state.
- Filtering incomplete TCP flows: This plugins scans a set of trace files and removes all incomplete TCP flows present. In a first scan, the connection tracking plugin is used to establish and update state for each flow found in the trace, before a second scan checks each packet's flow for complete three-way handshake and connection teardown. If those were observed, the packet is kept, otherwise it is dropped. The plugin can be accessed from the command line using `lndtool -r tcp-filter <trace1> [<trace2> <...>]`.
- Traffic Statistics: To quickly get an idea of what is contained in a trace file, we have developed a simple traffic analyzer plugin that computes counters and percentage values for the number of packets and bytes contained, IP payload protocol usage, TCP/UDP port number usage, and TCP flows. The plugin can be accessed from the command line using `lndtool -r traffic-analyzer <trace1> [<trace2> <...>]`.
- Abstract Protocol Header Definition: `libnetdude` and Netdude allow the developer to make protocol analyzers arbitrarily smart and to design the visual representation of protocol header data in any way desired. For example, the standard IP plugin is able to check whether the IP header checksum is correct, and can also fragment and reassemble packets. Frequently however, sophisticated functionality is less important than basic understanding of the protocol structure. In this situation, it is not necessary to force developers to write their own code to make a protocol's structure accessible. Rather, a high-level *protocol header definition language* is desirable that allows the specification of a protocol header's layout in a simple text file.

  We have designed a language, PHDL, for this purpose and provide an interpreter as a separate `libnetdude` plugin. When `libnetdude` is initialized, the PHDL plugin reads all installed protocol definition files and creates protocol data structures accordingly, equipping each new protocol with a header blueprint. When a packet's protocols are analyzed, the header blueprint is used to build an instance of the structured protocol data that can then be queried and manipulated. The complementing PHDL Netdude plugin then uses this information to visualize the protocol data in a standardized tree view similar to `ethereal`, but with the added functionality of being able to modify the header fields.

  As an example, we show a PHDL definition for IPv4 in Figure 9.

```
# PHDL Definition for IPv4 based on RFC 791.

# structure of the header common to many IP options.
def "opthdr" {
        unsigned int "type" 8;
        unsigned int "length" 8;
}

# structure of an IPv4 address -- 32bit field, when output,
# chunk into 8bit units and separate using a "."
def "ip4addr" {
        int "addr" 32 { unit = 8; sep = "."; }
}

# structure of IP options that contain a list of IPv4 addresses.
def "addropt" {
        opthdr "header";
        unsigned int "ptr" 8;
        chain "route" {
                ip4addr "addr";
        } until length ((.header.length - 3) * 8);
}

# Now the main header definition:
proto "IPv4" (net : 0x800) {

        block "fixed" {
                int "version" 4;
                int "hl" 4 { scale = 4; }

                block "tos" {
                        enum "ecn" 2 {
                                0 : "-";            1 : "ECT(0)2";
                                2 : "ECT(1)";       3 : "CE";
                        }
                        enum "tos" 4{
                                0x10 : "Low Delay"; 0x08 : "Reliability";
                                0x04 : "Low Cost";  0x00 : "None";
                        }
                }

                unsigned int "len" 16;
                unsigned int "id"  16;

                block "frag" {
                        int "rf" 1; int "df" 1; int "mf" 1;
                        unsigned int "off" 13;
                }

                int "ttl" 8;

                # The exclamation mark identifies this field as the
                # key to the selection of the next protocol header:
               !enum "proto" 8 {
                        1       : "ICMP";
                        2       : "IGMP";
                        4       : "IPIP";
                        6       : "TCP";
                        # rest treated as "other"
                }

                hex "checksum" 16;
                ip4addr "src";
                ip4addr "dst";
        }

        chain "options" {
                union "option" {
                        int "noop" 8 if (.noop == 0);

                        block "security" {
                                opthdr "header";
                                unsigned int "s"   16;
                                unsigned int "c"   16;
                                unsigned int "h"   16;
                                unsigned int "tcc" 16;
                        } if (.header.type == 130);

                        addropt "lsrr" if (.header.type == 131);
                        addropt "ssrr" if (.header.type == 137);
                        addropt "rr"   if (.header.type == 7);

                        block "streamid" {
                                opthdr "header";
                                unsigned int "id" 16;
                        } if (.header.type == 136);

                        block "timestamp" {
                                opthdr "header";
                                unsigned int "ptr" 8;
                                unsigned int "oflw" 4;
                                unsigned int "flag" 4;
                                ip4addr "addr";
                                chain "ts" {
                                        unsigned int "tstamp" 32;
                                } until length ((.header.length - 3) * 8);
                        } if (.header.type == 68);
                }
        } until length (fixed.hl - 5) * 4 * 8;
}
```

Figure 9: PHDL code describing the IPv4 header layout.

Other potential applications include traffic anonymizers, address mappers, import and export filters for other file formats, and interfaces to other software for more advanced functionality like visualization and mathematical analysis.

## 4 Real-world Use Cases

The original catalyst for the creation of Netdude was our work on TCP/IP network traffic normalization [HKP01]. This was a typical scenario for small-scale editing. In order to test our normalizations, we needed to create very specific packet constellations, for example specific values for the IP TTL field, the TCP flag bits, and IP fragments with valid and invalid fragment offsets. Using the Netdude GUI, we gave individual packets the desired features and replayed the manipulated trace files through the normalizer.

The second use case was in the domain of high-speed network monitoring equipment. The subject of study was Nprobe, a scalable multi-protocol network monitor [MHK+03]. The goal was to evaluate system performance under various traffic loads. We used `libnetdude` to create traffic patterns that triggered different hotspots in the system. We then wrote an IP address mapping plugin for `libnetdude`, that maps those traces to disjunct IP address ranges so that we could replay multiple instances of the traces in parallel to expose the probe to high volumes of traffic.

At the moment we are using Netdude in order to test intrusion detection system (IDS) signatures. The classic approach is to experiment with a signature for a network-based IDS [Pax98][Roe99], testing whether the IDS reacts correctly when replaying a trace file. It is often more straightforward to manipulate the traffic itself and not the signature, particularly when testing the resilience of a new signature against variation in traffic patterns and corresponding false positive rates. This approach was particularly useful in our work on the Honeycomb IDS signature generator[Kre03], where the ability to make small-scale modifications to packet data was most helpful for testing the string-matching algorithm used by the system. Netdude's editing capabilities have worked very well in these scenarios.

## 5 Discussion & Future Work

In its current state, we find the framework useful for everyday work with sets of trace files of sizes ranging from a few kilobytes up to several gigabytes. The ability to access functionality through a command line interface is most valuable for scripting tasks for repeated execution. We mostly use the GUI application for the simpler editing tasks and for quick inspection of trace files. Netdude's ability to handle large trace files makes it a far better option than alternative tools like `ethereal` that lack this feature and that are restricted to files smaller than the system's physical memory capacity.

When using the command line, we have frequently found that it would be useful to be able to use the traditional UNIX approach of piping the output from one processing stage to the input of the next stage. Unfortunately this metaphor is not directly applicable to our problem setting: depending on the functionality provided, a stage may have to employ random access to various locations in the file, or scan a file repeatedly (as in the case of the TCP filtering plugin described in Section 3.3). Directly piping packet data from one stage into the next will not work here since the streams cannot be rewound. However, temporary files could be used transparently, and the piping could be kept within the `lndtool` command, for example using a syntax like `lndtool '-r <stage1> -i <input> | -r <stage 2> | ... | -r <stage n> -o <output>'`

Another useful feature would be the ability to use `libnetdude` in a scripting environment such as Python or Perl. Creating the necessary "glue" code using a tool like SWIG[5] should prove fairly easy and is one of our next items for future work.

## 6 Summary

Netdude is a framework for inspection, visualization, and modification of `tcpdump` packet trace files. Its modular design allows users to interact with the framework at different abstraction levels: a low-level trace navigation wrapper for `libpcap` called `libpcapnav`, a high-level API with convenient types for performing common packet manipulation tasks in `libnetdude`, and a GUI application that allows both small- and large-scale editing previously impossible without writing code. The framework is readily extensible at the `libnetdude` and GUI levels through its plugin architecture, making it a workbench for the creation of new packet trace tools. A number of plugins have been developed so far and have already helped us in cutting down the development time for new features.

The system has been in development for three years. The use cases that allowed us to apply the framework so far have confirmed our goals of simplifying the development of packet manipulation code and encouraging the re-use of components developed in other projects. We have implemented a number of plugins for purposes such as IP address translation, TCP flow demultiplexing, and statistical analysis.

We hope that the authors of networking code consider using the Netdude framework for their future packet manipulation needs, and provide useful functionality in the form of plugins for `libnetdude` or the Netdude GUI as a benefit to the community. Netdude is provided with a BSD license, hosted on SourceForge, and can be obtained at `http://netdude.sf.net`.

## Acknowledgments

Since October 2002, this work has been carried out in collaboration with Intel Research, Cambridge. We would like to thank Vern Paxson, Mark Handley, and Jon Crowcroft for inspiration and helpful feedback. We also thank the Netdude user community for valuable ideas, comments, and contributions, particularly Andrew Moore, Daniel Stodden, and Euan Harris, who also provided valuable comments on the paper. Thanks also to the Castle Pub in Cambridge for hosting our brainstorming sessions.

## Notes

[1] See `http://www.tcpdump.org`

[2] See `http://www.ethereal.com`

[3] See `ftp://ftp.ee.lbl.gov/tcpslice.tar.Z`

[4] The implementation of this feature is significantly complicated by the fact that `tcpdump`'s packet analyzer is currently not available as a library.

[5] See `http://www.swig.org`

## References


[HKP01] Mark Handley, Christian Kreibich, and Vern Paxson. Network Intrusion Detection: Evasion, Traffic Normalization, and End-to-End Protocol Semantics. In *Proceedings of the 9th USENIX Security Symposium*, August 2001.

[Kre03] Christian Kreibich. Honeycomb - Automated NIDS Signature Generation using Honeypots, Poster Paper. In *Proceedings of ACM SIGCOMM 2003*, Karlsruhe, Germany, August 2003. SIGCOMM.

[MHK+03] Andrew Moore, James Hall, Christian Kreibich, Euan Harris, and Ian Pratt. Architecture of a network monitor. In *Passive and Active Measurement Workshop Proceedings*, pages 77–86, La Jolla, California, April 2003.

[Pax98] Vern Paxson. Bro: A System for Detecting Network Intruders in Real-Time. *Computer Networks (Amsterdam, Netherlands: 1999)*, 31(23-24):2435–2463, 1998.

[Roe99] Martin Roesch. Snort: Lightweight Intrusion Detection for Networks. In *Proceedings of the 13th Conference on Systems Administration*, pages 229–238, 1999.

# Trusted Path Execution for the Linux 2.6 Kernel as a Linux Security Module

Niki A. Rahimi
*IBM Corporation*

## Abstract

The prevention of damage caused to a system via malicious executables is a significant issue in the current state of security on Linux operating systems. Several approaches are available to solve such a problem at the application level of a system but very few are actually implemented into the kernel. The Linux Security Module project was aimed at applying security to the Linux kernel without imposing on the system. It performs this task by creating modules that could be loaded and unloaded onto the system on the fly and according to how the administrator would like to lock down their system. The Trusted Path Execution (TPE) project was ported to the Linux kernel as a Linux Security Module (LSM) to create a barrier against such security issues from occurring. This paper will attempt to explain how Trusted Path Execution is implemented in the Linux kernel as an LSM. It will also describe how TPE can prevent the running of malicious code on a Linux system via a strategically placed hook in the kernel. The usage of a pseudo-filesystem approach to creating an access control list for users on the system will also be discussed. The paper will further explain how TPE is designed and implemented in the kernel. This paper will show how the access control list is utilized by the module to place checks on the execution of code on the system along with a check of the path the code is being run in. Further, the origins of the "Trusted Path" concept and its origination in the OpenBSD operating system will be discussed along with how TPE was introduced to the Linux security community. The paper will conclude with a synopsis of the contents and future paths and goals of the project.

## 1 Introduction

The running of malicious code on a Unix system either via an active attacker or unsuspecting local user is one of the greatest threats to computer security we have these days. There are and have been several solutions to this problem. Many of the solutions only solve the problem at one layer of the system such as the network. The problem can and should be approached by applying security at various layers of the system. Much of the scenarios involving malicious code can be attributed to malevolent users placing damaging executables in an unprotected system and running them either remotely or via an unsuspecting local user. Other scenarios involving these unsuspecting local users occur when said user has written potentially damaging code themselves but are unaware that this is the situation. The innocent user attempts to run this buggy code and finds that they have caused major damage to their own system. The Trusted Path Execution LSM attempts to prevent such occurrences from happening.

The problems that malicious executables can cause are varied but for the most part the biggest issue is malicious code being placed on the system either intentionally or accidentally. There are several scenarios of how this can be bad for the system. If you can think of a good way to hose a system, it could probably be done with a malicious executable. The problem is greatly enhanced by systems that are unprepared for such attacks. Of course, where the executable resides and who is running it will be a major factor in how much damage it can do. Malicious code can be especially problematic in unsafe directories of a system where the parent directory of the malicious executable in question is world and/or group writeable. The malicious user is able to run code that when given the privileges of write can potentially overwrite or damage proper code on the system.

The Trusted Path Execution module attempts to prevent the problems that malicious code attacks create. It does so by protecting the system at the point from which the execution of a file takes place on a given system. The module patches the Linux kernel in such a way as to make a quick check of the current user's credentials and verifies that the executable is not being run in a vulnerable path on the system. If a situation is found where the module does not accept the security of the given situation, a failure will occur within the module and an error -EACCES will be returned.

The module utilizes a kernel hook within the Linux Security Module (LSM) framework in which it makes a check at exactly the point of execution on a file in the system. This check will verify whether the path is, in TPE's sense, "trusted" and whether the current user attempting to execute the file is also "trusted". If a situation is found where both the user and path are considered untrusted, then execution fails. All other scenarios will result in the execution being allowed.

The determination of whether the path and user are trusted is also made by the module. It will check whether the parent directory of the file attempting to be executed is root owned and whether it is group and oth-

er writeable. If it is root owned and neither group nor world writeable, the path is considered trusted. If the current user attempting to run the code is either root or is listed in a trusted user access control list, they are considered trusted. The combination of checking the user and path for trustworthiness will determine whether the executable will be allowed to run. If neither are trusted, execution will be denied by the TPE module.

The remainder of this paper will attempt to further explain the problems associated with running malicious code on a vulnerable system. It will also show how TPE proposes to solve this problem and will be concluded with a wrap up of the paper in whole and future thoughts on the subject.

## 2 Malicious Executables in Untrusted Paths on a Vulnerable System

Preventing the execution of malicious code is a fundamental component of ensuring a Linux system's security. The problem seems quite evident but the scenarios will vary and are rather difficult to anticipate. This is indeed one of the most important problems a system administrator will face when contemplating the security of their system at a local level. The point at which security must be applied to a system is another issue to consider. Protecting at the network layer does not guarantee those that are on the inside, that is to say normal users, won't do something to also jeopardize the system. Bad code is bad code, no mater where it comes from. Similarly, protecting against specific types of well known malicious code won't protect against newly created ones. Of course, having these protections are important. The more points of vulnerability protected the better.

What is malicious code? It is a very general description of any kind of software that can cause damage to a system. This software includes viruses, worms, backdoors, Trojan horses, etc... What exactly can occur? One of several things, but in general buffer overflows from faulty code, exploit programs that can override root access, erasure of core system files, overwrite of system files and so on. In many cases the system will be rendered useless. There are several ways malicious code can be placed in a system. Thus, given the scope of this definition, the problems created by malicious code are immense and should be of high concern to any system administrator.

Take for example, a computer virus. A computer virus is generally defined as a program or piece of code that is loaded onto a computer without the knowledge of the owner/admin and is run against said owner/admin's permission. This is a very broad definition and accordingly the problems associated with computer viruses are also very broad. The computer virus is thus one of the most important security issues these days. Once a computer virus is detected, the damage is normally complete and the system administrator may only have hindsight to their benefit. The administrator is now capable of preventing the specific virus from occurring again, but is still vulnerable to other viruses that have yet to be applied to the system. To make the problem even more difficult to solve, sometimes the case where malicious code is applied to the system is performed by a local user. In fact, many situations of harmful code being run on a system are initiated by a local user with absolutely no idea that their program is about to wreak havoc on their own system. In this scenario, the local user has written code that is defective.

What can make this malicious code even more effective and/or malicious is the location in which it resides on a system. If the directory under which the code resides in is group or other writeable, we have allowed for further situations where the code could potentially overwrite other programs/executables on the system. When the directory allows for group and/or other writeable permissions in particular, this is leaving the code not only especially vulnerable. Unless otherwise given user-only access, the file will default to receiving permissions that allow just about anyone else write access.

Similarly, if all users on the system are given equal access to running the code we open the system up to further attacks. The greater the number of users who are able to run code, the greater the chance of faulty or malicious code getting executed. Why should everyone be given such access, if only a few users actually need to run executables?

Thus, the problem of malicious code is far-reaching. It is a problem that is highly difficult to solve. There are many solutions to preventing such code from entering a system via the network, but once it gets in the system and there are no protections for the system itself. The code is now able to be run and cause the damage. The only surefire approach to preventing any kind of code to get into the system is to simply unplug it from the network. And what about the problem with local users? If the system is simply unplugged from the network, there is still the problem of the innocent local user with buggy code. Must the administrator also remove the users? At this point, an unplugged systems with absolutely no users except administrator serves very little purpose except for being really, really secure. This is where security solutions like TPE can come in and help.

## 3 How the TPE blocks malicious code from running on the system

Solving the problems associated with malicious code is quite difficult. The question of where and how the code gets implemented is important. Answering this

question is a great challenge. Simply containing a system from the network will not solve this problem, either. Common users that either want to cause problems or have no clue of their code's malicious intent will equally break a system. An approach that is both comprehensive and isolated in impact on a system must be taken. This approach must take in to account the various scenarios without making the system obsolete to normal users. It must also accept the fact that code that can harm the system will somehow make its way in. Indeed, it should know that the "bad" code is already in the system, just waiting to be run. With this in mind, the solution must find a way to minimize the chances of this "bad" code from being run without preventing the "good" code from execution. TPE aims to be exactly this type of solution.

TPE attempts to prevent users from accidentally executing malicious code by ensuring that only code installed by trusted users is permitted to execute. TPE is a Linux Security Module which enhances the security of the Linux kernel by monitoring the running of executables in "trusted paths" on a system by particular users. TPE accomplishes this by manipulating a strategically placed hook in the kernel that monitors the execution of files. It performs a check of the path in which the executable resides and the user who is attempting to execute the program. The check of the path will determine whether it is trusted or not. A "trusted path", in the TPE sense of things, is one in which the parent directory of a file is owned by root and is neither group nor other writable. The component of the trusted path that allows for root owned directories is a convenience to the system administrator as they should be able to actually run system critical code. As a result, unless otherwise altered, base Unix directories like /bin and /usr/bin are considered trusted, but /tmp is not (Phrack 53-08 and 54-06)[2,5].

Why do we trust a "trusted path"? We will consider such directories as already protected by certain features in which the environment the executable resides in does not allow for code to cause major damage. In what way? Well, we'd like to be able to execute code as root and hope that root did not allow Joe user either ownership or execution rights to such code as would be found in /usr/bin. Thus, /usr/bin is a "safe" environment for code to be executed. On the other hand, if Joe user happens to have code X in /joedir where either he has world or group write access, he will be able to run code X. If code X is faulty, either purposely or not, Joe has the potential to cause major damage to the system. As a result, we will consider /joedir an "untrusted path".

TPE makes use of a Trusted Users access control list to define "trusted users". Users placed on the list are considered trusted and will be able to run executables they normally have access to run on the system without intervention from TPE. Upon attempt to execute, TPE will check whether the current user attempting to run an executable is trusted or not. If Joe decides to create buggy and/or malicious code X in an untrusted path he does have access to, he will find that he cannot run the code due to the presence of TPE on the kernel. In this case, if Joe is a legitimate user on the system, he can request to be added to the trusted user list, which allows him access no matter where he runs the code. Thus root has choices in whom to allow privileged execution rights. This also minimizes the amount of users who will be able to run executables at a given time. Root must actively add users to the trusted list (root is already in the list upon module instantiation). If Joe-user only needs to login to the system and check his email, root can choose not to put him on the list. In this case, root is able to add those users who absolutely need to run code on the system and thus keep the trusted list to a minimum.

The administrator will pick only those users that critically need to run code. In addition, the administrator could choose to allow certain users access at certain times when, perhaps, the administrator has been able to review the code that needs to be executed before it is run. He/she may also decide to revoke execution access to those users who have jeopardized the system before. This gives the super user much more flexibility in controlling the actions on the system.

There are only four scenarios that can be evaluated, of course; trusted user/trusted path, trusted user/untrusted path, untrusted user/trusted path and finally, untrusted user/untrusted path. The first three scenarios will be allowed execution. In the scenario where an untrusted user in an untrusted path attempts to run an executable on a TPE-ified kernel, the operation will be prevented from occurring. In this case, TPE has successfully accomplished its goal of preventing the potential malicious code from doing its damage. The following table describes the scenarios:

| **User\Path** | **Trusted** | **Untrusted** |
|---|---|---|
| Trusted | Execution Allowed | Execution Allowed |
| Untrusted | Execution Allowed | Execution not allowed |

If malicious code is presented to the system from a remote machine, there is no way in which TPE can prevent this. Similarly, if an innocent user on the system accidentally writes buggy code, the module is not going to be able to do anything about it. What TPE takes into account the fact that malicious code can get into the system very easily but once it's on the system, it does not allow for it to cause the type of damage it would like.

TPE provides a line of defense to the system. It takes into account that malicious code will be on the

system in some form or other. It minimizes the entry points for this code to be run and thus minimizes the amount of scenarios that could cause an attack from such programs. The TPE module does not prevent such situations as a malicious user acquiring the root password and/or utilizing a suid attack. The module will not run a firewall or audit the system. TPE is one of several approaches to containing the system and making it more secure. It should be used alongside a good firewall and other beneficial forms of security. It should not be the only method of hardening a Linux system.

## 4 How TPE is implemented

The Trusted Path Execution is implemented into the kernel as a Linux Security Module [12]. The Linux Security Module framework [10] is made up of a set of hooks into the kernel that can be manipulated in various ways. Of course, the main purpose of the framework is to manipulate the kernel into becoming more secure but several of the hooks can be utilized for other utilitarian purposes as the module needs. The module must also make use of a sysfs pseudo-filesystem [6] that allows for user space to system space interaction. This is a replacement to the common use of command line/system call user to system interface and will be explained further below. The module had been previously created utilizing the sys_security call and had to be migrated to the new method. Until a new system call can be made available to the LSM project, the module will continue to use the pseudo-filesystem approach.

The name of the sysfs pseudo-filesystem that TPE utilizes is "tpefs". It represents a file system interface that presents a file that lists trusted users. A user is considered "trusted" if their uid is on the list. The TPE-ified system will verify whether a user is trusted by reading this list. For convenience, the trusted list can be manipulated by utilizing userspace write actions, such as "echo", to add and remove users from the list. It can also be utilized to show the list to userspace via userspace read commands such as "more". The list is created in memory upon instantiation of the module.

The core code of the module, tpe.c, relies on two checks upon the running of an executable in the system. Within the module this is accomplished by utilizing the tpe_bprm_set_security hook, which is always called upon file execution. The two checks are performed by two functions, TRUSTED_PATH(current path, current uid) and TRUSTED_USER(current uid). The TRUSTED_PATH() function verifies whether the path is root owned and whether it is either group or world writeable. The TRUSTED_USER() function verifies whether the user running the executable is listed in the tpe_acl trusted user list. The functions are called within the module and performed in the tpe header file, tpe.h.

### 4.1 The Access Control list and Pseudo-File system "tpefs"

The tpe_acl trusted user list is created upon initialization of the module from a call to the tpe_init() function. In order to modify the tpe_acl list a sysfs pseudo file system [11] called "tpefs" is also created by the module. Two files are created in the "tpefs" filesystem which when written to, actually edit the tpe_acl list in memory. Thus, the filesystem is really not a file system but a method by which adminstrator can send information from user-space to kernel-space. This is why we call it a "pseudo" filesystem.

By default, root or uid 0 is added to the tpe_acl list upon initialization of the module. Root is protected from deletion from the list by a check in the code. Other users must be added utilizing the "tpefs" file system. Similarly, removing users from the list is performed with a "del" file. Both files, "add" and "del" are created for the filesystem by default from within the module code. The two file approach was utilized rather than a single file in order to keep the code and administration of the module simple for both the kernel and the user.

It should be noted that the usage of the sysfs pseudo-filesystem approach as opposed to a normal system call and command line method was due to the recent drop of the sys_security system call. There were a few other modules that were affected by this, including DTE[1] and SELinux[14]. Both projects are now utilizing the pseudo-filesystem method, as well. This seems to be the standard method by which the modules will be accessing system space from user space.

In order to instantiate the tpefs filesystem, the kernel must be compiled to include the LSM patch and the tpe.c module must be chosen to be installed as a module. Once both actions are taken, a partition must be mounted as type sysfs, as follows:

```
mount -t sysfs sysfs /<mountpoint>
```

Next, the module must be inserted into the kernel via the insmod command: insmod tpe.o. At this point, a subdirectory under the sysfs mount point, <mountpoint>, created above, will be created under the name tpefs. There will be two files created by the module, namely add and del. In order to add a user, one simply needs to perform a write operation on the "add" file in the following manner:

```
echo <uid> > <mountpoint>/tpefs/add
```

In a similar, deleting a user will involve a write to the "del" file is performed with the following command:

```
echo <uid> > <mountpoint>/tpefs/del
```

Notice that utilizing the "echo" command is just a suggestion. Any other write command will work in a similar manner. Using "echo" is probably the simplest choice and is the one preferred by our team.

The "add" file may also be utilized to show the list and a description of this is given further on in this section. This is performed by performing a read action

on the "add" file. This will instantiate a copy to be made of the list in to a user space buffer. This buffer information is then sent to stdout as a list of uids. The code is implemented from within the module by the trustedlistadd_read_file function(struct file *file, char *buf, size_t count, loff_t *offset) . Upon issuing a command such as cat <mountpoint>/tpefs/add, the list of uids will be presented to stdout. Currently, the maximum number of users that can be added to the TPE access control list is limited to 80. As more work is performed on the module, this will be altered to allow for a greater number of users and/or be made dynamic.

The code only allows for numeric user ids to be added on the command line at this point. It does return an error if any non-numeric values are entered. It also checks whether any other odd combinations are given as uids and returns an error. If any duplicate ids are attempted to be added , the module will also return with an error. This is similar for the removal of user ids. Any errors will be logged in the kernel info log for the system administrators to be able view. This adds a sort of auditing feature of the module and is found throughout the TPE codebase.

In addition to the tpe.c file, the module includes a header file called tpe.h. This is the only other non-document file associated with the module. It contains various macro definitions and a few sub-routine functions for the module. Most importantly, the TRUSTED_PATH() and TRUSTED_USER() functionality. The tpe_init function is also highly important:

Some the lesser involved macros in the tpe.h file include the TPE_ACL_SIZE value, which sets the size of the tpe_acl array to 80and the ACK, NACK and DUP macros which specify return values for the subroutines. Two important functions to the tpe_acl list are defined in the file, tpe_verify and tpe_search. The tpe_verify function will search for the uid that is currently being attempted to be added to the tpe_acl by the administrator. The result of this function will tell the TRUSTED_USER() function whether the uid is valid or not. The tpe_search is utilized by tpe_verify to determine if the uid is already on the list. If the uid is on the list, an error is returned so that repeat uids are not added to the list. These macros are all highly important to the module and are useful in making it more refined and user-friendly.

A documentation file for tpe called tpe.txt is also available on the LSM patch. The document is installed along with the module in the /Documentation/lsm directory. This document will give the user information about the module, how to install it and contains guidance for utilizing the tpefs pseudo-filesystem.

# 5 Evaluation

The actual effects the Trusted Path execution module makes to the system have been shown to be unintrusive, safe and effective. The module does in fact secure the system in the way it says it should. It is compatible with the Linux kernel and causes no performance issues. To be more accurate, the module has only been run on 2.5/2.6 Linux kernels and therefore this information is only pertinent to those particular kernels.

## 5.1 Security

The module has been thoroughly tested for functionality. It does do what it says it should and that is to enhance security by blocking execution of files. The four basic scenarios of trustworthiness were applied to a system and the results were evaluated. These four scenarios were: trusted user/trusted path, trusted user/untrusted path, untrusted user/trusted path and untrusted user/untrusted path. The results of these tests concluded that execution was allowed to be performed only in the first three scenarios listed above. It was also shown that when an untrusted user in an untrusted path attempted to execute a file, it was denied. This was the crucial point that TPE was indeed applying its security policy to the kernel.

As an example, let's say user Joe would like to run executable "buggy" in the /tmp directory. Prior to loading TPE on the system, so long as Joe has execute permissions to the file, he will be able to execute "buggy" from within the /tmp directory. Joe then goes on to wreak havoc on the system, eventually bringing it down. The administrator is now left with an obsolete system and must reinstall. If the administrator could have turned back the clock and installed the Linux Security Framework on his kernel along with enabling the TPE module, he/she would have saved themselves a headache.

If TPE had been installed on this system and the trusted list was created without Joe's userid added, Joe would have attempted to run "buggy" and found that execution would have been denied. Upon Joe running "./buggy", TPE's hook in the kernel, tpe_bprm_set_security hook, is called in. No matter what form of execution takes place on the system, TPE will always be called and do the check of the path and user. TPE has detected that Joe is not a trusted user and that /tmp is not root owned and neither group or other writeable. In other words, TPE is checking whether the user and/or path are "trusted". Since both user and path or not trusted, in this case, execution is denied by the module and -EPERM is returned to stdout. At this point "buggy" was not run and the system is still up. TPE has saved the day.

Suppose the system administrator wanted Joe to run "buggy" for some sadistic reason. In order to allow Joe permission to run his "buggy" executable, the administrator need only add Joe's userid to the trusted list.

Once Joe is on the list, his attempts to run "buggy" will be allowed. Perhaps the adminstrator needed to crash the system.

Consider the scenario of an innocent user seeking code from outside the system. The TPE module will not be able to block a user from downloading code from an external resource, such as an ftp download web site or outside host. If this code is malicious/buggy, the system has no way of finding this out. But this does not indicate TPE has failed. Once the code is on the system, it is not able to perform its malicious intent because TPE has blocked it from being executed.

It has been shown that via various testing models, the above scenarios indeed occur with and without the Trusted Path Execution LSM. These tests would check kernel kernel log messages for the appropriate logs from the TPE module. These were all found to be cleanly applied to the kernel event log and were appropriate to the actions taking place during the testing scenario. The tests would also verify that no errors were being logged by the kernel that were not expected at the given action of the TPE module within the system.

## 5.2 Compatibility

The code has been thoroughly reviewed by both the IBM and LSM communities. As a result of this review several enhancements have been made to make the code to make it more effective. We have also modified the code to make it less capable of becoming buggy. One enhancement that came out of the code review was adding kernel spin locking capabilities to make the module smp safe. Another important upgrade for the module was moving from the pcihpfs pseudo-filesystem approach to the sysfs pseudo-filesystem. The move to the new sysfs filesystem made for a cleaner code base. The code was also greatly reduced in size and much easier to debug as a result. This migration also reduced the amount potential vulnerable spots in the code. Thus, once again the code is much less capable of becoming harmful to the kernel and/or operating system.

TPE has been implemented as a Linux Security Module and accepted by the LSM community. Therefore, it abides by the framework the LSM community utilizes to attach to the Linux kernel. It has been written according to guidelines set by this community. The LSM community in turn works closely with the Linux kernel development community. As a result, the module also adheres to coding rules and styles as set by the Linux kernel community. The LSM framework is scrutinized by the Linux kernel core development team as well as the lkml mailing list and anyone else interested in the kernel. Thus, the project has several eyes scrutinizing it. Most importantly, it is scrutinized by those especially knowledgeable with the Linux kernel and system security.

The Trusted Path Execution LSM is still considered new and experimental and thus should be thoroughly reviewed as it progresses as a module. It is indeed simple and small enough that it is not anticipated to be a great effort to test its value to the kernel. The module was written with security coding standards being a high priority and will continue to be modified with this consideration in mind. The module was run through several tests including system and functional verification. These are described in more detailed below.

The TPE LSM has been thoroughly system tested. Basic testing was performed to verify that the module's implementation in the kernel does not break other programs and is cleanly applied to memory. As mentioned in the Security section above, kernel logs were verified so that no inappropriate events occurred during the loading, unloading and run-time actions of the modules. No testing was performed while other security modules were loaded, as TPE was not created to be a stackable module. If at some point in time, this is not the case, stackability will also be tested.

The trusted user list that is created in memory was thoroughly tested. Users were added and removed from the list while the actual list was monitored. It was shown that the list was effectively manipulated and accurately reflected the desires of the system administrator as to who should be listed. Invalid values, nonuids, were attempted to be passed to the list for addition and deletion. The module appropriately denied addition of these invalid parameter values. In addition, uids that were not on the list were attempted to be removed. The module was able to recognize that these uids were not on the list and acted appropriately with an error return. Overall, manipulation of the tpe_acl trusted list was deemed accurate and clean of problems/bugs.

Memory tests were also conducted on the tpe_acl list to verify that no overwrite of memory would take place during actions on the list. Several of the tests were attempts upon overloading the tpe_acl array with too large or too many uid values. Checks in the code prevented this from occurring. As this was the only parameter the module creates in to memory, it was the only object that needed this sort of testing.

The LSM project is also constantly under test and scrutiny. In fact, a fellow IBMer, Trent Jaeger, has created a set of projects to verify the LSM project [9]. Trent and his team are working towards verifying that the LSM framework appropriately implements the security actions of each module. They have used the CQUAL static analysis tool to make sure every security relevant operation could be controlled by an LSM hook. They also created a runtime analysis tool, Vali [3], that finds inconsistencies in operations authorized by the modules. The discoveries that Trent and his team found utilizing their analysis tools have led to improvements of the LSM along with validation that the

framework can indeed be utilized to secure the system.

Once TPE is instantiated on the system, there are some limitations to keep in mind. The system is locked down and simple execution is controlled, thus ordinary users may find some annoyances to deal with when wanting to run executables but they don't have access rights to do so because of the module's presence on the system. A TPEified system will essentially become more of a "governed state", where greater interaction must occur between the end user and system administrator(s).

Although we have performed several tests on this module, it should be noted that this is no guarantee that all potential bugs/defects have been removed from the code. It should also be noted that this is still an experimental project and more time and effort will improve the validity of the code. Further testing is always necessary and shall be performed in the future as new kernel levels are introduced and features are enhanced on the module itself.

## 5.3 Performance

No specific performance testing was completed on the Trusted Path Execution module. Upon instantiation of the module, no visible performance impact was made and thus benchmarking was not deemed necessary. The module makes use of only one kernel hook and thus is quite small on its impact into the system. The usage of the "tpefs" sysfs filesystem is created upon instantiation of the module, as are several other smaller components of the module. Given that the "tpefs" codebase is the largest portion of the module, once we are past insmod'ing the module, there is very little code that actually augments the kernel. If performance benchmarking is deemed necessary in the future for TPE, it will be performed at that time.

# 6 Related work

The meaning of the "trusted path" has been previously defined [13] by a much older concept in security.

According to the "Secure Programming for Linux and Unix HOWTO (see url below), "A trusted path is simply some mechanism that provides confidence that the user is communicating with what the user intended to communicate with, ensuring that attackers can't intercept or modify whatever information is being communicated." This definition applies to a context where security is to be tested on a particular system and is not quite associated with the concept of a "trusted path" as defined in the Trusted Path Execution Loadable Security Module project. The original idea presented a similar concept in that the original ensured that the login prompt was legitimate. This is similarly to how the Trusted Path module ensures that all programs an untrusted user executes are legitimately put there by root. For further information see http://www.dwheeler.com/secure-programs/Secure-Programs-HOWTO/trusted-path.html.

The second definition of a "trusted path" was defined in a project that the TPE LSM was based on. Trusted Path Execution project was originally created for OpenBSD 2.4 as a direct patch to the kernel by Mike Schiffman. It was described in Phrack 52-06 and later modified by the Stephanie project [15] for OpenBSD 2.8 and 2.9. The patch was distributed under the two clause BSD license. The usefulness of this project as an enhancement to Unix security was recognized by the Linux Security Module community and subsequently suggested as a potential module. There are several differences between the original BSD patch and the LSM version. The most notable difference is, of course, in how they are implemented into the BSD and Linux kernels, respectively. The Stephanie project also brought in a few more checks into the BSD kernel, such as restricting symbolic links via the Openwall project from Linux. The Stephanie project also uses an actual system call to implement a tpe_adm command to modify the TPE trusted user access control list, whereas the LSM utilizes the sysfs pseudo-filesystem. Despite these major differences in code, the core concept of the Trusted Path Execution kernel check is the same in all of the projects mentioned above.

There have been a handful of other Linux Security Modules that have been implemented into the Linux kernel prior to the Trusted Path Execution module. The SELinux LSM is one that is most notable. It makes use of the Linux Security Module framework to implement the Flask mandatory access control architecture model into the kernel. It is an example of a much larger implementation of the LSM framework. The project is led by Stephen Smalley and Peter Loscocco of the National Security Agency. The Domain and Type Enforcement (DTE)[7] LSM is another module of note. Created by fellow IBMER, Serge Hallyn[4], it makes use of a mandatory access control model that assigns types to files and domains to processes and furthers this idea by associating which domains can access which types. Interestingly enough, both module implementations, SELinux and DTE, were initially direct Unix kernel patches prior to becoming LSMs, much like the Trusted Path Execution module. Both DTE and SELinux are currently available via the LSM patch. The SELinux module is also available directly on the 2.6 kernel along with the LSM patch.

The Linux Security Module project [8] is continuing its efforts to create a comprehensive set of security modules for the Linux kernel. It is maintained regularly and current patches are available for download off of their main web site. According to it's site maintainers, LSM "provides a lightweight, general purpose framework for access control"(see lsm.immunix.org). The

project was created as a means of providing security to the Linux kernel without adding the overhead of direct patches to the kernel. The user may choose which module to implement and thus have greater control of the security of the kernel. The usage of the module approach makes the task of securing the system much more flexible and dynamic. There are several modules to choose from at this point and a couple more have been added since the introduction of the TPE LSM. Since the project is fairly new to the kernel, many of the latest modules are still considered experimental. There are several well established modules, including SELinux, DTE and owlsm. These projects are all current and actively maintained.

## 7 Conclusion

The Trusted Path Execution LSM has been designed to enhance the security of the Linux 2.6 kernel. In its ability to prevent the running of malicious executables, the module shows one way that the kernel can be manipulated in order to protect a system from potential damage. It is blind to whether the malicious code was intentionally created or not, and thus covers both scenarios. By performing a check into the kernel at the point of file execution, the module is able to monitor whether the path the executable resides in is "trusted "and whether the user is considered trusted. If both the path and user are "untrusted", the module will prevent execution from occurring.

TPE was accepted by the LSM community in May of 2003. The module was submitted under a dual BSD/GPL license. TPE was integrated into the mainline LSM BK tree immediately and placed on their official patch to the 2.5.70 kernel. The current version of the TPE module has been available on all LSM kernel patches since then. It is anticipated to be placed in the official 2.7 kernel once development commences on that project. TPE is a fairly new addition to the LSM lineup.

Further enhancements to the project will be added in the future to increase the value-add the module can bring to a system. This may include increased administrative capabilities and further checks into the filesystem. It is anticipated that the module will be of great use to many system administrators seeking to improve the security on Linux.

## Bibliography


[1] Lee Badger, Daniel F. Sterne, David L. Sherman, Kenneth M. Walker, Sheila A. Haghighat. Trusted Information Systems, Inc. A Domain and Type Enforcement UNIX Prototype. 5th USENIX Security Symposium. June 1995. Salt Lake City, Utah.

[2] Krzysztof G. Baranowski. Linux Trusted Path Execution Redux. Phrack 53-08. July 1998.

[3] Antony Edwards, Trent Jaeger, Xiaolan Zhang. Runtime Verification of Authorization Hook Placement for the Linux Security Modules Framework. ACM Computer and Communications Security. November 2002. Washington, D.C.

[4] Serge Hallyn, Domain and Type Enforcement for Linux. http://www.cs.wm.edu/~hallyn/dte.

[5] routel daemon9. Hardening the Linux Kernel (series 2.0.x).Phrack 52-06. January 1998.

[6] William von Hagen. Migrating to Linux kernel 2.6. LinuxDevices. Com article. February 2004

[7] Kenneth M. Walker, Daniel F. Sterne, M. Lee Badger, Michael J. Petkac, David L.Shermann, Karen A. Oostendorp. Confining Root Programs with Domain and Type Enforcement(DTE). 6th USENIX Security Symposium. June 1996. San Jose, California.

[8] Chris Wright and Crispin Cowan. Linux Security Modules: General Security Support for the Linux Kernel. USENIX Security Conference. August 2002. Ottawa, Ontario.

[9] Linux Security Analysis Tools. http://www.research.ibm.com/vali/

[10] Linux Security Modules. http://lsm.immunix.org

[11] LWN article. Avoiding sysfs surprises. June 2003.

[12] Security modules begin to appear. LWN Article on TPE. May 2003. http://lwn.net/Articles/31571

[13] Secure Programming for Linux and Unix HOWTO. http://www.dwheeler.com/secure-programs/Secure-Programs-HOWTO/trusted-path.html

[14] SELinux Documentation http://www.nsa.gov/selinux/info/docs.cfm

# Modular Construction of DTE Policies

Serge E. Hallyn
*IBM Linux Technology Center*
*Austin, TX 78759*
hallyn@cs.wm.edu

Phil Kearns
*College of William and Mary*
*Williamsburg, VA 23185*
kearns@cs.wm.edu

## Abstract

This paper describes a tool which composes a policy for a fine-grained mandatory access control system (DTE) from a set of mostly independent policy modules. For a large system with many services, a DTE policy becomes unwieldy. However, many system services and security extensions can be considered to be largely standalone. By providing for explicit grouping, namespaces, and globbing by namespaces, inter-module access rules can be made generic enough to permit modules to be mixed and matched as needed. As a result, it becomes easier to extend a policy, debug a policy, and to distribute meaningful policy modules with new software.

## 1 Introduction

Domain and Type Enforcement (DTE) [1] is a fine-grained mandatory access control system. An implementation exists for Linux as a Loadable Security Module (LSM) [16]. The DTE LSM reads the policy it enforces through a text file through sysfs. The policy language closely resembles TIS's original DTEL policy language, which was explicitly intended to be intuitive to read and write. We have previously presented tools to analyze and edit policies. We now present a tool to compose policies from policy modules, which are smaller, simpler policy excerpts. In practice, we find policy modules far simpler to work with than a single large policy.

We begin by describing DTE and DTE policies in more detail. Next we describe the syntax of a policy module. We describe methods of grouping types and domains, the priority assigned to access rules based upon source and target, and hooks for system interaction during policy compilation. Then we describe a ftpd protection policy previously presented[7], and show how the ftp-relevant portion of this policy becomes a module.

## 2 DTE

DTE specifies two types of labels, called types and domains. It assigns types to files, and domains to processes. File access is controlled from domains to types, and signal access is controlled between processes in different domains. A process may transition to a new domain only on a call to `execve`. A domain may only be entered through files labeled with types explicitly marked as entry types for that domain. These files are called entry points for that domain. Domains may only transition to certain other domains. There are two types of domain transitions. The first, called `exec`, is voluntary, while the second, called `auto`, is mandatory. When a process under some domain executes a file which is an entry point to another domain, to which the first domain has auto access, then the process will transition to the new domain. If the file was an entry point to another domain to which the first had exec access, then the process ordinarily does not switch domains. It may request a domain transition by performing

```
echo -n <new_domain> > \
      /proc/<pid>/attr/exec
```

before executing the new file.

The DTE policy file specifies all types, all domains, the file system's type assignments, domain to type access, domain signal access, permitted domain transitions, and domain entry points. See [7] for more details about the policy file.

Policies for fine-grained MAC systems are mostly constructed as one unit and by hand. For instance, a massive effort is under way to create a complete, safe, SELinux policy for several distributions of Linux [13, 14]. Tools exist [8] for analyzing DTE policies, and such work is also being done for policies of other fine-grained MAC systems [9, 12, 15]. Nevertheless, working with a large policy remains a painful experience. However, when working with large policies, patterns begin to emerge. Policies typically consist of several sets of domains and types. The entities within a set work together to achieve some goal, and the sets often interact very little. For instance, in the ftp policy presented in [7], the domain `ftpd_d`, and the types `ftpd_t` and `ftpd_xt`, work together to protect the system from an unsafe binary. By removing these entities, and all references to them, the remaining policy becomes simpler. We call this collection of domains, types, and all access rules pertaining to them, a module. The ftp module is shown in Figure 3, and will be described in Section 4.

Allowing policies to be composed from simple, meaningful, and coherent pieces will serve several purposes.

First, creation of policies will become far more efficient. For instance, when adding a new domain to an existing policy, one might have to enter hundreds of type accesses in order to get it properly interacting with the current policy. In contrast, modules allow domains and types to be grouped at several levels, and access to be specified using any of these groups.

Second, adding a feature to a policy, such as a new method of controlling access to the shadow file, or protection from a critical binary in which an as-yet unsolved vulnerability has been found, will become a simpler task. The module can be written entirely from its own point of view. Furthermore, in researching the state of the current policy, in order to understand how to properly insert a new feature, one need only look at those modules which can affect the new functionality.

Third, modules may be helpful in simplifying the analysis, and proof of invariants, of policies. For instance, several modules may be trivially shown to be irrelevant to the ability of the `inetd` daemon, if remotely exploited, to erase the `utmp` log file.

Finally, because a module generally encodes domains, types, and access rules which work together toward some end, it is a natural way to express the security policy changes necessary for a new piece of software. Software companies and free software groups, therefore, could distribute policy modules along with software packages.

## 3 Module File Specification

We now discuss the structure of a module file. The module syntax specification follows.

```
<module_file> ::=
  <module>+

<module> ::=
  Module <mod_name>
    [<domain_def>|<type_def>| \
    <group_def>]+
  end

<domain_def> ::=
  domain <dom_name>
    <dom_line>+
  end
```

A module file may contain more than one module. Each module may contain several domain, type, and group definitions, as well as the access rules pertaining to them.

```
<dom_line> ::=
  entries <type_name>+  |
  [absolute] signal [in|out] <gen_dom> \
      <sig_num>  |
  [absolute] domain [in|out] <gen_dom> \
      [auto|exec|none] |
  [absolute] type <gen_type> <type_acc>|
  assert <policy_name> <data> |
  DEFAULT_DOMAIN
```

The domain definitions declare a unique name for the domain, a set of entry types, and a set of access rules pertaining to the new domain. Domain transition or signal access rules may be `in`, in which case they specify access from other domains to the new domain, or they may be `out`, defining access from the new domain to other domains. Since types are passive objects, which cannot themselves access other types or domains, the type access rules in a domain definition do not include the `in` or `out` keyword.

Exactly one domain definition applied to a policy must contain the keyword `DEFAULT_DOMAIN`. That domain will be assigned to the first process on the system.

```
<type_def> ::=
  type <type_name>
    <type_line>+
  end

<type_line> ::=
  <path_type> <path_name>+ |
  [absolute] access <gen_dom> <type_acc>|
  <default_type>  |
  assert <policy_name> <data>

<default_type> ::=
  DEFAULT_ETYPE | DEFAULT_UTYPE | \
      DEFAULT_RTYPE
```

Type definitions declare a unique name for the type, a set of paths assignment rules, and a set of access rules. Clearly, the access rules are only incoming from domains. A type whose definition contains the keyword `DEFAULT_RTYPE` will be assigned to the root of the file system, and recursively to its descendants until another type assignment rule applies. Alternatively, one type may be labeled as `DEFAULT_ETYPE`, and another may be labeled as `DEFAULT_UTYPE`. The first will be assigned to the root of the file system, and the second will be assigned to its descendants until another type assignment rule applies.

Both type and domain definitions may contain `assert` statements. These are used for maintenance of policy constraints. They are stored with the type definition until module application, but their interpretation and enforcement is defined by the named policy consistency class, any number of which may be written by the policy authors to ensure the maintenance of any module properties. The last line of the `ftpd_xt` type definition in Figure 3 is an example of an assert statement,

instructing a module loaded as `blp` to label this type as protected.

```
<group_def> ::=
  group domain <dom_name>
    import <dom_name>+
  end

<group_def> ::=
  group type <type_name>
    import <type_name>+
  end

<gen_dom>  ::= all | none | <dom_name>
<gen_type> ::= all | none | <type_name>
```

Grouping is accomplished on several levels. First, the keyword `all` refers to all domains or types which are currently known. Second, a group definition in a module may bind a name to a set of domains or types.

For instance, the following module segment defines a group of domains which may transition to user domains, and may require to files such as `.bashrc` and `.xsession`.

```
group login_domains_g
        import login_d su_d
end
```

A separate `x11` module might extend this group using

```
group login_domains_g extend
        import xdm_d
end
```

in order to borrow `login_d`'s and `su_d`'s rights to read user login files.

The following module segment defines a type which is actually called `root_t`.

```
type base.extraneous.root_t
        DEFAULT_RTYPE
        [...]
end
```

Since `root_t` is the type name which will be used in the final DTE policy, no names within the namespace may actually clash. Modules may refer to this type using any of the following names:

1. `all`
2. `base.+`
3. `base.extraneous.+`
4. `base.extraneous.*`
5. `base.extraneous.root_t`
6. `root_t`

In addition, any type groups which have imported this type can also be used to refer to this type.

The name `base.extraneous` may be a real type, or it may simply be a namespace placeholder, depending on whether any module defines a type by that name. A namespace placeholder is the parent of a domain, type, or group, which is not itself defined to be a domain, type, or group. It can be referred to during namespace globbing, but will not appear in the final policy.

| Type of access | Priority level |
|---|---|
| Absolute single in | 12 |
| Absolute single out | 11 |
| Absolute group in | 10 |
| Absolute group out | 9 |
| Absolute all in | 8 |
| Absolute all out | 7 |
| Single destination in | 6 |
| Single destination out | 5 |
| Group in | 4 |
| Group out | 3 |
| Default (all) in | 2 |
| Default (all) out | 1 |

Figure 1: Priorities of access rules

Namespace globbing works as follows. When a name ends in `.+`, it refers to all descendants under this name. When a name ends in `.*`, it refers to only the immediate children of this name. Therefore `base.+` includes `base.extraneous.root_t`, but `base.*` does not. If `base.extraneous` were itself a type, then `base.*` would include this type, as would `base.+`.

## 3.1 Priority of Access Rules

Since domains and types can declare conflicting access rules, we must clearly define the priority of access rules. Much thought has been given to the current priorities, which have been somewhat modified following experience with an earlier module compiler prototype. The priority takes the form of an integer between 1 and 12. The priority assigned to access rules is shown in Figure 1.

Each type of access consists of three pieces of information. First, it can be `in` or `out`. This is relative to the type or domain in which it is defined. When a domain specifies a certain type access, this is a `out` access rule, as the access is outbound from the domain. If a type defines access from some domain, this is `in`, as the access is inbound from the domain to the type. The second piece of information relates to the precision of the rule target. When an access rule names a specific domain or type, this is `single` access. If the rule names a group, or a namespace expansion such as `Services.*`, this is `group` access. If the rule targets the keyword `all`, this is of course `all` access. Finally, the rule is either `absolute` or not. This depends only upon whether the

access rule is preceded by the keyword `all`.

If two conflicting rules have been defined pertaining to the access permitted from a domain to another domain or type, then the rule with the highest priority will be applied. For instance, the base module's definition of type `base_t` specifies that all domains have `absolute` access `rxld` (read, execute, lookup, and descend) to `base_t`. This rule is `absolute all in`, and therefore has a priority of 8. Assume we write a new module, defining a domain intended to contain untrusted code. The domain definition might contain the statement:

```
absolute type all none
```

This rule is `absolute all out`, and therefore is priority 7. Since an `absolute all in` access rule has a higher priority than `absolute all out`, the new untrusted domain will receive `rxld` access to `base_t`, even though it asked for none. Had it in fact gotten none, then it would not be able to access any types at all, as it could not descend to them through the root of the file system. Similarly, if any types defined in the new module are intended to be accessed by the untrusted domain, then these types must specify incoming access from the untrusted domain as `absolute`, to ensure that that it will override the untrusted domain's outgoing type access definition. On the other hand, the base policy specifies a type `bin_t`, which includes a normal `group in` definition. As this is of a lower priority than `absolute out`, the access rule specified by the new module's untrusted domain is chosen, denying the untrusted domain all access to type `bin_t`. As we will see, this is a crucial element of the ftp module, preventing the ftp server from providing attackers with root shells, for instance.

Note that incoming access overrides outgoing access for the same target precision and `absolute` status. More specific rules override more general rules, unless the `absolute` keyword is present in one of the rules.

The usage of these keywords is intended to be intuitive. However, a switch to usage of simple numeric priority has not been ruled out. For instance, in place of

```
absolute domain in \
login_domains_grp auto
```

a module would specify

```
domain in login_domains_grp \
auto 60
```

The disadvantages to this are that module authors might require a deeper understanding of how policy compilation is affected by the priorities, and would need to consider these effects explicitly for each access rule.

## 3.2 Module Application

A set of modules may be applied simultaneously, and more than one set may be applied in series. For instance, we may begin by combining a set of base modules, then apply a set of service modules, and finally apply a module to ensure a particular security feature. We must therefore clearly define the behavior of group expansion across multiple module applications.

For named domain and type groups referenced in access rules, the group is expanded at the time of module application. In other words, for each member of the group, a new access rule is defined with the same access details as the original rule. Each newly created rule is associated with a `group` priority, to ensure proper resolution of any future conflicts. If the group has not yet been defined, an error is raised and compilation fails. For namespace globbing, that is, * and +, the currently defined descendants and children (respectively) of the parent being expanded are used. For instance, assume we applying a module which contains the rule

```
domain some_domain
      type base.exec.+ rwx
end
```

If the only children of `base.exec` defined thus far are the two types `base.exec.sbin` and `base.exec.bin`, then only these types are included in this rule. A later module may define type `base.exec.javabin`, but this type will not be added to the access rule.

The `all` target keyword is treated somewhat differently. An access rule directed at `all` will be expanded at the time of module application. Again the new access rules resulting from the expansion are stored with an `all` level for later conflict resolution. However, a generic form of the rule is also stored. All such generic rules are expanded each time a set of modules is applied. If the rule had not previously been applied, any policy consistency modules will be consulted at the new rule creation, just as with any other new access rule. For example, the base module defines default access `rld` to type `base_t` for `all` domains. This rule is expanded after each module application, so that all domains will be granted this access.

## 3.3 Keyword Substitution

One of the goals listed in Section 2 for the use of policy modules is to facilitate distribution of policy modules with new software. It must therefore be possible to apply policy modules across a variety of systems. To accomplish this in any meaningful way will often require some bit of system interaction. For instance, a policy module distributed with `xdm` might require labeling each user's `$HOME/.xsession` as an entry type

to the user domain. This requires system interaction to determine valid users on this system who actually have a `$HOME/.xsession` file.

The prototype module compiler provides system interaction through an `exec` keyword. This is augmented with looping support over variables which have been set using `exec`. Using these features, an excerpt of the xdm policy might look as follows:

```
1  define xsession_f exec /bin/ls \
2    /home/*/.xsession
3
4  type xdm_out_fromuser_et
5    epath /etc/X11/xdm/Xsession
6
7    foreach file 'xsession_f'
8      epath 'file'
9    endforeach file
10
11   access user_d rwxlcd
12   access login_domains_grp r
13 end
14
15 domain user_d extend
16   entries xdm_out_fromuser_et
17 end
```

The first command, on lines 1 and 2, assigns to the variable `xsession_f` the result of executing the command `/bin/ls /home/*/.xsession`. This will contain a list of all user `.xsession` files, one per line. Lines 7 through 9 loop over each line returned by the `ls` command, each time adding a new epath line to the `xdm_out_fromuser_et` type definition, and replacing `'file'` with the next file. The result is a type to which the user domain may write, and which those domains which are members of the `login_domains_grp` group may read. The last three lines extend the `user_d` domain such that other domains may transition into it by executing the `.xesssion` files which were found. Of course, in many cases more complicated calculations than a directory listing will be required. The output from any script or program can be assigned to variables. However, the use of complicated external scripts might add an unwelcome element of unpredictability to the policy creation process. The policy consistency classes will offer some support to system administrators trying to keep this in check, and graphical analysis tools will remain available for analyzing the final policy.

### 3.4 Inheritance

An issue which may deserve further consideration is that of inheritance. It would seem to make sense to construct the type namespace such that certain properties, perhaps `absolute` access rules, are automatically inherited by the children of a type. On the other hand, this may simply needlessly complicate the process of policy creation, the simplification of which is the precise goal of the policy compiler. Currently, the notion of inheritance does not exist in the module compiler.

## 4 Ftpd Protection Module

Ftp daemons provide a great deal of interaction, usually with completely unauthenticated, or anonymous, users. In order to permit user logins, however, some ftp daemons run as root. A programming error such as buffer overflow or string format vulnerability can therefore lead to the execution of arbitrary commands using superuser privileges by anyone on the internet.

### 4.1 Original Policy

Figure 2 demonstrates policy to protect a system from ftpd. While a DTE system could actually boot and run with this policy, it is a minimalist policy designed only to protect from ftpd. An actual useful policy would be much larger, but contain a nearly identical set of ftp protections. The policy provides protection from attackers by containing the ftp daemon to a domain, called `ftpd_d`, which has limited access rights. This domain is not allowed to transition into any other domains, so that any code executed (legitimately or not) by the ftp daemon will also be subject to the same access restrictions. The domain is automatically entered whenever a privileged process executes `/usr/sbin/in.ftpd`. It requires permission to execute its entry point, library files, and files located under `/home/ftp/bin`. It needs read and write access to devices, `/home/ftp/incoming`, a transfer log, and some temporary files. The domain is refused the ability to execute anything it might have written. It has permission to read under `/home/ftp`, `/etc`, and, unfortunately, the password and shadow files. However, it lacks permissions to execute files under `/bin`, `/usr/bin`, etc. Therefore all existing exploits, which require the ability to execute "`/bin/cat /etc/passwd`" or `/bin/sh`, will fail.

### 4.2 Ftp Module

We now separate the ftp functionality out from the policy and into a module. The ftp module is found in Figure 3. It again defines a `ftpd_d` domain, and `ftpd_t`, `ftpd_et`, `ftpd_xt`, and `ftpd_wt` types. The `ftpd_d` definition specifies inbound domain transitions from `boot_t`, and from all domains defined under `Admin.services`. `Ftpd_d` may not transition to any other domains, so this access rule is `absolute`. This does not completely rule out `ftpd_d` being permitted to transition to another domain. However, in order for `ftpd_d` to be allowed to transition to another

```
# ftpd protection policy
types root_t login_t user_t spool_t binary_t lib_t passwd_t shadow_t dev_t \
      config_t ftpd_t ftpd_xt w_t
domains root_d login_d user_d ftpd_d
default_d root_d
default_et root_t
default_ut root_t
default_rt root_t
spec_domain root_d (/bin/bash /sbin/init /bin/su) (rwxcd->root_t \
      rwxcd->spool_t rwcdx->user_t rwdc->ftpd_t rxd->lib_t rxd->binary_t \
      rwxcd->passwd_t rxwcd->shadow_t rwxcd->dev_t rwxcd->config_t \
      rwxcd->w_t) (auto->login_d auto->ftpd_d) (0->0)
spec_domain login_d (/bin/login /bin/login.dte) (rxd->root_t rwxcd->spool_t \
      rxd->lib_t rxd->binary_t rwxcd->passwd_t rxwcd->shadow_t rwxcd->dev_t \
      rxwd->config_t rwxcd->w_t) (exec->root_d exec->user_d) (14->0 17->0)
spec_domain user_d (/bin/bash /bin/tcsh) (rwxcd->user_t rxwcd->shadow_t \
      rwxcd->spool_t rxd->lib_t rxd->binary_t rwxcd->passwd_t rwxd->root_t \
      rwxcd->dev_t rxd->config_t rwxcd->w_t) (exec->root_d) (14->0 17->0)
spec_domain ftpd_d (/usr/sbin/in.ftpd) (rwcd->ftpd_t rd->user_t rd->root_t \
      rxd->lib_t r->passwd_t r->shadow_t rwcd->dev_t rdx->ftpd_xt \
      rd->config_t rwcd->w_t d->spool_t) () (14->root_d 17->root_d)
assign -u /home user_t
assign -u /tmp spool_t
assign -u /var spool_t
assign -u /dev dev_t
assign -u /scratch user_t
assign -r /usr/src/linux user_t
assign -u /usr/sbin binary_t
assign -e /usr/sbin/in.ftpd ftpd_xt
assign -r /home/ftp/bin ftpd_xt
assign -e /var/run/ftp.pids-all ftpd_t
assign -r /home/ftp ftpd_t
assign -e /var/log/xferlog ftpd_t
assign -r /lib lib_t
assign -e /etc/passwd passwd_t
assign -e /etc/shadow shadow_t
assign -e /var/log/wtmp w_t
assign -e /var/run/utmp w_t
assign -u /etc config_t
```

Figure 2: A DTE policy to protect from *wu-ftpd*.

```
Module Service.ftp
     domain ftpd_d
          entries ftpd_et
          absolute domain out all none
          domain in boot_d auto
          domain in Admin.services.+ exec
          absolute type all none
          signal out boot_d 14,17
          signal out Admin.services.+ 14,17
     end

     type ftpd_t
          access all none
          absolute access ftpd_d rld
          rpath /home/ftp
     end

     type ftpd_et
          access all r
          absolute access ftpd_d rx
          epath /usr/sbin/in.ftpd
     end

     type ftpd_xt
          access all none
          absolute access ftpd_d rxld
          access root_d rwcld
          rpath /home/ftp/bin
          assert mblp protect
     end

     type ftpd_wt
          access all none
          absolute access ftpd_d rwcld
          rpath /home/ftp/incoming
     end
End
```

Figure 3: FTP Policy Module

domain, the other domain would have to explicitly ask for `ftpd_d` to be permitted to transition to it, or add `ftpd_d` to a group and provide that group with inbound transition access.

Type `ftpd_t` is located under `/home/ftp`. Only `ftpd_d` may observe this type, no one may modify or execute. The file `/usr/sbin/in.ftpd` is the entry type through which `ftpd_d` may be entered, signified both by the `ftpd_et` type definition, and the `entries` line in the `ftpd_d` definition. The files under `/home/ftp/bin`, labeled as `ftpd_xt`, may be executed by `ftpd_d`, and written by `root_d`. There is no single domain which may both modify and execute these files. Finally, the files located under `/home/ftp/incoming`, labeled `ftpd_wt`, may be written, but not executed by `ftpd_d`. It may not be accessed by any other domains.

This set of accesses was also accomplished using the ftp policy. In fact, the module will eventually be compiled into a policy. However, using the module, we are able to limit statements concerning `ftpd_wt` to the 6 simple lines which define the type, and trust that any domains which are later added under `Admin.services` will be able to transition to `ftpd_d`.

A detailed discussion of

```
assert mblp protect
```

is beyond the scope of this paper. However a brief explanation is appropriate. If no policy constraint class named `mblp` has been loaded, then this line will be ignored. If this class has in fact been loaded, then it is instructed to label this type, `ftpd_xt`, using the keyword `protect`. A class may do with this information what it likes. It will be called once before and once after each application of a set of modules, and given a copy of the policy at each point. The `mblp` class, in particular, will print a warning if any domain is in fact allowed to modify the protected type. This demonstrates the simplest use of policy consistency classes. We could in fact write a class to simply read assertions which must hold true in the policy. More interesting classes, such as `mblp`, compare calculations on the policy before and after module application.

The most significant advantage of separating the ftp module out from the base policy becomes apparent when we consider writing more modules. For instance, the base policy module does not allow users to change their passwords. To add this functionality, we use a module such as that in Figure 4. Nothing in the ftp module needs to change, and we do not need to consult the ftp module while writing the password module. In contrast, adding password functionality to an existing policy could become very invasive.

# 5 Control

The simplest way to compile DTE modules into a policy is to use the command line utility `dte_pc.py`. A list of the modules to be applied is placed into a file, which is given as a command line argument to `dte_pc.py`. The resulting policy is placed into a file also specified as an argument.

Using `dte_pc.py`, all modules are applied simultaneously. Greater control over module application can be had on the python command line, or by writing custom module application scripts. The following python lines, for instance, combine the two modules `base` and `user`, and then apply a third module, `ftp`.

```
from DTEModule import ModuleFile
# firstmods will be an array containing
# the "base" and "ftp" modules
firstmods = ModuleFile("base").Modules()
firstmods.extend(
       ModuleFile("user").Modules()
)
ftpmod = ModuleFile("ftp").Modules()
p = DTEPolicy.Policy()
# Apply "base" and "user" together
p.apply_modules(firstmods)
# Now apply "ftp" separately
p.apply_modules(ftpmod)
p.write("dte_output_file.conf")
```

One advantage of using this code is the enhanced precision in group definitions as described in Section 3.2. That is, if any groups are defined and referenced in `base` or `user`, and then extended in `ftp`, then the references to them in `base` and `user` will not include the members added in `ftp`. Additionally, policy consistency classes are only invoked before and after each DTEPolicy `apply_modules()` invocation, so the above code would force the application of `ftp` to be more closely scrutinized. If all modules are applied at once, then a policy consistency class will only ever compare an empty policy to the final policy, which may not be useful, depending upon the policy consistency class.

# 6 Related Work

The policy language read by the DTE module is based in large part on the DTEL policy language used by the original DTE on Unix implementation [1]. The policy consistency classes and related assert statements are a generalization of the ideas proposed in [2]. Here Bell-LaPadula [4] and Strict Integrity [3] relations, assured pipelines [5], and the Clark-Wilson [6] concepts of constrained data items (CDIs) and transformation procedures (TPs) were used to guarantee maintenance of certain properties through dynamic policy changes. OO-DTE [11] applied DTE to CORBA distributed objects, and introduced an object oriented policy language,

```
Module password
     # domains passw_d
     # types passw_t passw_et shadow_t
     type passw_et
          epath /bin/passw
          access all rx
          access Admin.admins rwxlcd
     end

     type passw_t
          epath /etc/passwd /etc/passwd.tmp /etc/.pwd.lock
          access all r
          access passw_d rw
     end

     type shadow_t
          epath /etc/shadow
          access all none
          access login_domains_grp r
          access passw_d rw
     end

     domain passw_d
          type conf_t rlcd
          entries passw_et

          domain in all auto
          domain out all none
     end

End
```

Figure 4: Password Policy Module

DTEL++. In OO-DTE, a user's domain was used to determine permission to execute or implement methods assigned to particular types. This is quite different from the meaning of DTE in a operating system such as Linux.

SELinux policies [13], like DTE policy modules, are compiled, in this case to a binary policy file. The SELinux policy makes liberal use of macros, which are defined throughout the policy, and compiled using the `m4` preprocessor. SELinux policies are less structured than policy modules. There is no sense of domains and types being objects, of access rules belonging to the definition of the source of target of the definition, or of priority of conflicting access rules. SELinux policies make use of simple `assert` rules for safety constraints, but no attempts have been made to provide more in-depth analysis of the effects of a particular piece of policy during compilation. SELinux policies are more detailed and more complicated than DTE policies. The possibility and usefulness of transcribing the idea of policy modules to SELinux policies while keeping modules readable and small, remains to be investigated.

Tools exist to aid in editing and analyzing DTE and SELinux policies [8, 12, 15]. These tools analyze whole policies, and therefore complement, rather than compete with the DTE policy modules concept. The policy consistency classes used by the module compiler are designed to analyze the effect of particular policy enhancements on the overall policy. The existing DTE policy analysis tools can still be used on the policies resulting from module compilation.

IBM Research is investigating the concept of access control spaces [10], and working toward a method to determine whether an SELinux policy satisfies certain integrity goals [9]. This work again analyzes whole policies. Ultimately, it is possible that Linux vendors could use this approach to verify the correctness of a TCB included with their distribution, while system administrators could use policy consistency classes to analyze the effects of their own policy modules on the base policy.

## 7 Conclusion

By the very virtue of being fine-grained, policies for MAC systems such as DTE and SELinux become very large, currently tens of thousands of lines for SELinux. Policy modules break this into a number of smaller pieces, and permit authors to intelligently group objects and subjects to permit concise and expressive access rules. The careful construction of a policy module language results in a far more convenient, more efficient, and safer policy specification. In practice, it has greatly eased the movement by the authors between various testing and development machines with various distributions.

## 8 Availability

The DTE LSM is available as part of the LSM project at `http://lsm.immunix.org`. The policy compiler is available from `http://www.nekonoken.org`. Both are licensed under the GPL.

## 9 Acknowledgments

Hallyn's work was supported in part by a USENIX Scholarship. The authors also wish to thank the paper shepherd, Crispin Cowan, for his helpful and constructive comments.

This work represents the view of the authors and does not necessarily represent the view of IBM. IBM is a registered trademark of International Business Machines Corporation in the United States, other countries, or both. Other company, product, and service names may be trademarks or service marks of others.

## References

[1] Lee Badger, Daniel F. Sterne, David L. Sherman, Kenneth M. Walker, and Sheila A. Haghighat, *A Domain and Type Enforcement UNIX Prototype*, Usenix Security Symposium (1995).

[2] Tim Fraser and Lee Badger, *Ensuring Continuity During Dynamic Security Policy Reconfiguration in DTE*, Proceedings of IEEE Symposium on Research in Security and Privacy, 1998.

[3] K. J. Biba, *Integrity Considerations for Secure Computer Systems*, Mitre Technical Report ESD-TR-76-372, 1977.

[4] D. E. Bell and L. J. LaPadula, *Secure Computer Systems: Unified Exposition and Multics Interpretation*, Mitre Technical Report ESD-TR-75-306, 1976.

[5] W.E. Boebert and R.Y. Kain, *A Practical Alternative to Hierarchical Integrity Policies*, Proceedings of the National Computer Security Conference, 1985.

[6] David D. Clark and David R. Wilson, *A Comparison of Commercial and Military Computer Security Policies*, Proceedings of the IEEE Symposium on Security and Privacy, 1987.

[7] Serge Hallyn and Phil Kearns, *Domain and Type Enforcement for Linux*, ALS 2000.

[8] Serge Hallyn and Phil Kearns, *Tools to Administer Domain and Type Enforcement*, LISA 2001, p. 151-156.

[9] Trent Jaeger, Reiner Sailer, and Xiaolan Zhang, *Analyzing Integrity Protection in the SELinux Example Policy*, Usenix Security Symposium, 2003.

[10] Trent Jaeger and Xiaolan Zhang, *Policy Management using Access Control Spaces*, ACM Transactions on Information and System Security, 2003.

[11] Durward McDonnel and David Sames and Gregg Tally and Robb Lyda, *Security for Distributed Object-Oriented Systems*, DARPA Information Survivability Conference and Exposition, June 2001.

[12] MITRE Corporation, *SLAT: Information Flow Analysis in Security Enhanced Linux*, available at `http://www.nsa.gov/selinux`.

[13] Stephen Smalley, *Configuring the SELinux Policy*, NSA Technical Report, `http://www.nsa.gov/selinux/papers/policy2-abs.cfm`.

[14] Stephen Smalley et al, *SELinux mailing list*, `http://www.nsa.gov/selinux/list-archive/summary.cfm`.

[15] Tresys Technology, *Security-enhanced Linux Policy Tools*, `http://www.tresys.com/selinux/`, 2003.

[16] Christ Wright, Crispin Cowan, James Morris, Stephen Smalley, and Greg Kroah-Hartman, *Linux Security Modules: General Security Support for the Linux Kernel*, Usenix Security Symposium, 2002.

# Managing Volunteer Activity in Free Software Projects

Martin Michlmayr
*Department of Computer Science and Software Engineering*
*University of Melbourne*
*Victoria, 3010, Australia*

*Centre for Technology Management*
*University of Cambridge*
*Mill Lane*
*Cambridge, CB2 1RX, UK*

martin@michlmayr.org

## Abstract

During the last few years, thousands of volunteers have created a large body of free software. Even though this accomplishment shows that the free software development model works, there are some drawbacks associated with this model. Due to the volunteer nature of most free software projects, it is impossible to fully rely on participants. Volunteers may become busy and neglect their duties. This may lead to a steady decrease of quality as work is not being carried out. The problem of inactive volunteers is intensified by the fact that most free software projects are distributed, which makes it hard to quickly identify volunteers who neglect their duties. This paper shows Debian's approach to inactive volunteers. Insights presented here can be applied to other free software projects in order to implement effective quality assurance strategies.

## 1 Introduction

The success of free software in the last few years, including projects such as Linux and Apache, has clearly demonstrated that the free software development model is a viable alternative to proprietary software development. Free software is a matter of liberty, rather than price, since it offers the user more freedom than proprietary software, such as the freedom to run, copy, distribute, study, change and improve the software. One reason for the success of the free software development model, which is tightly connected to the Unix philosophy [19], is related to the open nature of free software: a large number of developers inspect the code and get involved in a project. Eric S. Raymond studied the free software development model in detail and published his observations in a widely acclaimed paper [18]. During his case study, Raymond found that there is a high amount of parallelization in the debugging process. Due to the open nature of the source code in free software projects, anyone can review the code, find defects and contribute bug fixes. Raymond suggested that this 'bazaar' model, in which a large number of volunteers review the code and contribute feedback and patches, is the reason for the success and high quality of many free software projects. This suggestion meshes well with findings in software engineering which show that peer review significantly contributes to quality [9].

Free software development is typically characterized by two factors. First, free software projects are carried out by volunteers. Second, the volunteers of a project are distributed. However, not all projects exhibit these two characteristics. Due to the success of free software, an increasing number of companies get involved in free software projects. Some companies allow their paid developers to get involved in distributed free software projects. Also, some software is being developed in a traditional fashion by paid programmers working in the same building and is then released as free software. This example shows that it is important to make a clear distinction between the way software is developed and the license under which it is distributed. A piece of software can be developed in a traditional way and then be licensed as free software. However, this paper covers a certain aspect of software developed according to the free software development model, that is, projects which are distributed and largely carried out by volunteers.

This free software development model has generated a large amount of very popular and successful software in the last few years. Due to the recent popularity of free software, there is an increasing number of academic studies and papers which concentrate on successful projects, such as Apache [16], GNOME [14] and Linux [20]. However, there are also a large number

of unsuccessful free software projects which have not yet attracted the attention of researchers. SourceForge hosts over 70,000 projects, but a large number of them are not actively developed. Since free software projects do not have budgets, there is usually not a specific point when unsuccessful projects get wrapped up. It is therefore hard to identify projects which have been failures. Nevertheless, it is clear that some free software projects fail or have severe quality problems.

The distributed and volunteer nature of free software projects make them unique in software development, and is related to certain advantages, such as the high amount of peer review mentioned previously. However, there are several drawbacks and challenges associated with the free software development model as well. Due to the volunteer nature of free software projects it is impossible to fully rely on participants [15]. Also, coordination and management can be very complex in distributed projects with volunteer participants. For example, it is impossible to put all participants of a distributed free software project in a room, give out tasks, and only leave the room when everyone agrees to perform the tasks they have been given.

Free software projects have to deal with these challenges and find solid solutions. The success of free software shows that many challenges have been overcome, and the development model is continuously refined as more experience is gained. One major challenge, which is becoming an increasing problem, is volunteers who suddenly stop carrying out their duties without informing others. In many free software projects, volunteers have specific responsibilities. While a large number of participants make infrequent contributions, there are some volunteers who play crucial roles in a project, and who therefore have to be constantly involved. For example, the Linux kernel has a number of trusted lieutenants through whom code submissions of specific parts of the kernel are carried out [17]. They are central to the development, since other volunteers cannot contribute if the trusted lieutenants do not carry out their work. Similarly, if the main developer of a very small project becomes inactive, the whole project may come to a halt. If such volunteers become inactive, especially without informing others so a backup can be arranged, projects face important problems. There are therefore two problems: a project can stop completely if a core developer becomes inactive, or the quality of a project may suffer.

It is therefore very important to observe and investigate the problem of inactive volunteers from a quality assurance perspective and to discuss possible solutions. In the following, I will approach this problem from the perspective of the Debian Project, and discuss mechanisms Debian has implemented to deal with this problem. While some solutions described in this paper are specific to Debian, many lessons can be learned from Debian's experience of dealing with inactive volunteers. It is my hope that members of other projects will gain a better understanding of the problem through this paper and map Debian's strategies to their respective projects.

## 2 Background

As discussed in the previous section, volunteers who are not performing their duties can have a significant impact on the quality of a project. Therefore, it is important to take this problem into consideration in a project's quality assurance effort. In this section, I will discuss and summarize the problem, and describe why it is such a big problem especially for Debian.

### 2.1 Inactive Volunteers

The motivation of volunteers in a free software project is different than that of paid developers involved in the development of a commercial application. Raymond has observed that developers get involved in free software projects to "scratch an itch" [18]. This explains why certain tasks are not carried out in free software projects. For example, most free software projects do not have a written specification, simply because writing one is a tedious job most volunteers are not interested in. Similarly, users of free software are not in a position to demand new features from the developers of a project. If they require functionality nobody else is interested in developing, they can get involved themselves and contribute to the project, or pay someone to implement the missing features.

There are two schools of thought about the responsibilities of a volunteer. One school maintains that a volunteer is free to do whatever they wish at any time and does not have any obligation at all. The other school of thought claims that once a volunteer has agreed to perform a specific function they have a responsibility to fulfill it and to tell others when they cannot perform their duties anymore. From a quality assurance perspective, it does not really matter what a particular volunteer thinks – certain measures have to be taken in *any* case. However, it is clear that it is easier to work with volunteers who have a certain diligence in their responsibilities. Ideally, a volunteer will realize when they can no longer perform their function, for example because they are too busy or have lost interest, and will arrange for a replacement or backup. Unfortunately, experience from Debian and other projects shows that this is often not the case.

The problem of inactive volunteers is tightly connected to the nature of the free software development model. The fact that most free software projects are performed by volunteers makes solutions fairly complex. Due to the volunteer nature of free software projects, it

is inevitable that participants become busy from time to time. Student volunteers have exams and participants who earn their living in companies might have less time sporadically when a project is due. For all participants it is true that personal circumstances may change and leave less time to contribute to free software projects. Furthermore, participants in free software projects can sometimes experience 'burnout' [12]. There are many variables that may change and affect the participation in a free software project. However, it is clear that earning a living and 'real life' take precedence when time is short. Hence, it is important that a free software project does not rely on a specific volunteer for a crucial task but instead implements redundant structures [15].

The problem is aggravated by the second characteristic of free software projects: its participants are distributed. This fact makes it very hard to identify the personal circumstances of a developer. In a traditional, commercial software project, people would notice very quickly if something happened to one of their developers. Their colleagues would wonder why a developer is not coming in to work. They would easily *see* that something is wrong because the developer is not present. Unfortunately, it is not as easy in free software projects where coordination and communication mainly relies on e-mail – there is not much one can do when a volunteer does not respond to inquiries by e-mail.

The combination of these two factors makes it very hard to identify inactive volunteers in free software projects. Moreover, the question whether a participant is active is of a gradual rather than a binary nature. If a volunteer has not performed their duties for a month, it is possible that they are just temporarily busy and will come back later. This makes it very hard to draw the line and make a decision when to take action. A volunteer might just be on holidays for three weeks, but one does not know that because it was not explicitly announced.

Raymond covered the problem of free software maintainers who do not perform their duties anymore briefly in his paper "Homesteading the Noosphere" [18]. It is a well-known problem and some guidelines have been discussed, such as when to fork or take over an abandoned project. There are also two terms which have been associated with inactive volunteers. *AWOL*, which stands for "Absent Without Leave", and *MIA*, short for "Missing In Action", are both used to refer to volunteers who become inactive without informing others.

While the problem is known, no solutions have been developed which are generally applicable. It is also a very delicate problem because one has to maintain the balance between respecting a volunteer for what they have done while at the same time pointing out that they are not performing their duties anymore and suggesting that it is time to move on. Before describing Debian's approach to the problem, an explanation of why inactive volunteers are a severe problem specifically for Debian is in order.

## 2.2 Debian

Debian is one of the most popular and certainly the largest Linux distribution available to date [11]. The mission of Debian is to provide a complete operating system comprising free software. While Debian develops some software itself, such as Debian's package manager `dpkg`, the main task of Debian is not to develop software. Instead, Debian obtains software from other sources (the so-called "upstream" developers) and integrates the different pieces of software into one system. The integration is done through the creation of a Debian package which complies to certain guidelines which are outlined in a document known as the Debian Policy manual [5]. Every package is under the control of one or more maintainers, and most volunteers in Debian maintain one or more packages. While there are some volunteers who help out with other tasks, such as maintaining the web site or the software archive, publishing security updates, or performing quality assurance functions, the majority of volunteers in Debian are package maintainers.

Although there are some packages which are being maintained as a team, the majority of packages are currently maintained by one person. As the sole responsibility for keeping a package up-to-date and free of defects rests on that package's maintainer, there exists a heavy reliance on those volunteers. Achieving more redundancy by having backup maintainers and maintainer teams has been suggested [15]. While there is a slow trend away from having a single maintainer for a package, the majority of packages are still maintained by one person. Therefore, there is a high liability to packages becoming unmaintained as volunteers become busy or entirely inactive. This mapping of one package to one single maintainer is the reason why inactive maintainers are such a huge problem for Debian. In some respects, Debian is like a bazaar of cathedrals: a project comprising a large number of individual projects of varying size, each with its own owner.

There is one additional factor which makes the situation very complex: Debian's growth. Debian has grown in size over the years, and especially in recent years, quite drastically (see figure 1). The enormous size of Debian makes it very hard to keep track of every maintainer. With over 800 maintainers, it is impossible to know each volunteer in Debian, and it is very hard to spot if one out of over 8,000 packages in the archive is not maintained properly. Also, while Debian as a whole has grown in size, the quality assurance team has remained about the same size.

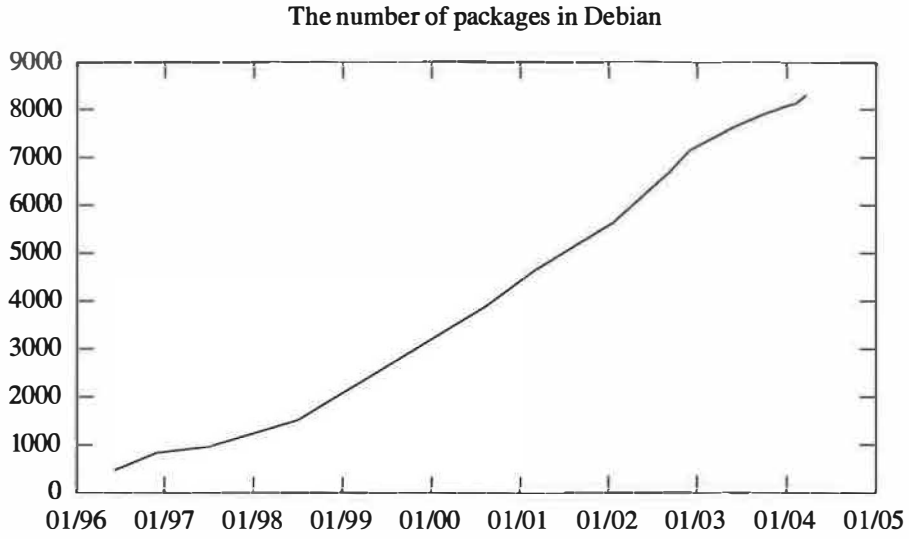



Figure 1: The size of Debian has steadily increased over the years. At the beginning of 2004, the project featured over 8,000 packages.

Taken together, these factors constitute a large quality assurance problem which has to be solved. In the following sections, I will describe ways to locate and track inactive maintainers, and show tools which have been developed to assist in these tasks. Before covering inactive maintainers, it is important to give some attention to the full life cycle of a maintainer and to discuss how one actually becomes a package maintainer in Debian.

## 3 Debian's New Maintainer Process

Over the ten years of Debian's existence, many processes have been adapted and refined in adherence to the ways in which Debian has changed. Debian's admission process was very informal in the early days. Everyone knew each other at that time. Prospective members could simply send an e-mail stating their interest and they would be added to the project. It was also very common to help out with the packages of another maintainer when they were busy. As Debian mushroomed, the processes slowly became more formal in order to deal with the large number of developers and the size of the Debian system.

As Debian reached a size where it was impossible to know every other volunteer, a new admission process was needed. The New Maintainer process [4] was introduced to handle new volunteers. Debian's New Maintainer process is much more elaborate than the admission processes of most free software projects. One reason for this is that every Debian developer has to be fully trusted. Since Debian packages are installed as root on a user's machine (that is, with full permissions), it is necessary to strictly control who can upload packages to the Debian archive. Each software package uploaded to the archive must be digitally signed with the GPG (GNU Privacy Guard) or PGP (Pretty Good Privacy) key of a Debian developer. When a volunteer fulfils all steps in the New Maintainer process and gets accepted as a Debian developer, their GPG key is added to the Debian keyring against which signatures of every package upload to the archive is verified.

The New Maintainer process is composed of three steps. First, every volunteer needs a GPG key which is digitally signed by an existing Debian developer. This involves a face-to-face meeting which allows performs a social function and intends to get new members better integrated. Second, the volunteer's philosophy with regards to free software and understanding of Debian's Social Contract [7] is checked. Third, the volunteer's technical skills are tested in various ways. Once an applicant has passed all three steps, a thorough report is sent to the Debian Account Manager (DAM) who decides whether an account is created. Debian's constitution grants the Debian Account Manager the authority to admit new people to the project or to remove existing developers [2]. Traditionally, the DAM has exercised the former right, but as inactive maintainers get identified it also becomes important to remove volunteers from the project.

The New Maintainer process plays an important role in the problem of inactive maintainers because it selects which volunteers are admitted to the project. In many large free software projects, it seems fairly typical to make a one-time contribution of source code. The project benefits from this contribution as the code

gets integrated in the project and is then maintained by others. The original contributor does not have to show a high level of commitment, as the code has been integrated and the contribution is therefor useful. However, in Debian few people are interested in contributing patches for old bugs or assisting other maintainers. Most volunteers are interested in maintaining packages on their own. For that, however, you need a constant level of contribution. Otherwise, the package is not maintained well.

The New Maintainer process is vital in ensuring that only volunteers who show a high level of commitment get admitted to the project. If large numbers of new volunteers who do not understand the problem they cause by neglecting their commitments are admitted, the problem is intensified. It is therefore important to tackle the problem at its root and educate new maintainers. It is important to take into account that the problem is largely a social one. In many cases, volunteers realize that they are busy, but they do not admit to themselves that they are actually *too* busy, and are harming the project. They often think they will find the time "tomorrow", but in reality it never happens. Hence, it is important to discuss this problem from the beginning, and the New Maintainer process is a good venue for this.

Over the last years, the New Maintainer process has become more elaborate and longer than in the past. While this change was not a conscious decision, I think this is partly because of inactive maintainers. If the admission process takes a long time and is quite complex, only those volunteers who are really interested will go through the process. Once they have cleared this hurdle, it is likely that they will stick around for a long time. The New Maintainer process therefore acts as a screening process not only to select volunteers with good technical skills, but also those who are likely to show a high level of commitment to the project in the future. However, this is only speculation on my part, and it is too early to tell whether this strategy really works out. It will be interesting to conduct a study in a few years to see whether the duration of the admission process is linked to the likelihood of a volunteer neglecting their duties. For this study, you would first investigate how long or complex the joining process was for someone, and then check whether they are still active a certain number of years after joining. Even without an elaborate study, it is clear, though, that the admission process has to take the problem of inactive maintainers into account.

# 4 Locating Inactive Maintainers

As mentioned before, finding volunteers who are neglecting their duties is much harder in a distributed volunteer project than in a company where everyone comes to work every day. In this section, I will describe the sources of information Debian takes into account when searching for inactive maintainers.

Due to the volunteer nature of free software projects, it is quite likely that some volunteers will neglect their commitments, especially in large projects. It is therefore important to take this possibility into account and to mention it in the developer's documentation. For example, the Debian Developer's Reference explicitly asks volunteers to inform other members of the project when they go on holiday and provides a section describing how to retire from the project properly [3]. This way, arrangements can be made so other volunteers take over their tasks, either temporarily while they are on holiday or permanently when a volunteer retires. Most communication in free software projects is based on e-mail or IRC (Internet Relay Chat, a form of real-time communication) and many participants have never met in person. While e-mail is usually a very effective means of communication, it completely fails if someone does not answer their e-mail – unfortunately, this is likely the case if someone is busy or something serious has happened. Contacting the person by phone is often a more effective alternative in this case. Hence, having the phone number of each volunteer and additionally also an emergency contact, either a relative or a friend, are very helpful in finding out what happened to a volunteer if they do not respond to e-mail. While Debian currently does not require either a phone number or an emergency contact in its developer database due to privacy concerns, participants will be able to enter this information on a voluntary basis in the future.

## 4.1 Sources of Information

In most free software projects, many sources of information are available which can be used in order to form a judgement whether a volunteer is inactive. In the following, I will describe sources typically used to locate inactive maintainers in Debian.

### 4.1.1 Echelon

Like most free software projects, Debian relies on electric means of communication. While IRC is used by a large number of volunteers in Debian, the primary infrastructure is based on e-mail. Debian has an automatic system which monitors every mailing list and looks for postings sent by a Debian developer. The system, known as Echelon, uses various means to determine whether a mail is sent by a Debian developer, such as recognition of the e-mail address or verification of the signature of a GPG or PGP signed message. When a message has been recognized as one posted by a Debian developer, an activity value is updated in Debian's internal LDAP database. This database is used to store various bits of information about every official Debian developer.

In addition to fields for name, login and other personal information, there is a field which indicates the time a message has been last posted on the Debian lists by a developer. This field also lists the Message-ID of the e-mail so it can be found easily. In addition to the activity value, there is a field which indicates when the last digitally signed message has been posted. Since all Debian uploads have to be signed, this activity field gives an indication when the maintainer has last uploaded one of their packages.

Debian's Echelon system is a tremendous resource during the search for inactive maintainers. While it is not 100% reliable, since it sometimes does not recognize a message to be from a Debian developer, it gives a very good first indication of whether a participant has been around recently or not. This system is very useful to see whether it is worth looking at other sources of information.

### 4.1.2 Package Information

Many sources of information are directly related to the job of a Debian maintainer – maintaining packages. If the packages of a specific maintainer appear unmaintained, this is a clear indication from a quality assurance point of view that certain measures have to be taken. While unmaintained packages do not necessarily imply that a maintainer is inactive, the situation has to be dealt with. It is sometimes the case that some packages are not maintained well because a maintainer is overwhelmed by the amount of work they have. In that case, it may be that they maintain their important packages well but neglect others which they deem to be less important. However, in some cases, unmaintained package are due to a maintainer simply neglecting all their duties.

There is a wide range of information which can be used to judge how well a package in the Debian archive is being maintained.

**Release Critical Bugs**: Debian has a Bug Tracking System through which bugs can be reported [1]. Each bug has a severity, ranging from minor to critical. Debian defines bugs of the severities serious, grave and critical to be "release critical", meaning that a package with bugs of that class is not considered to be ready for release. The Bug Tracking System provides an easy overview of all bug reports of a specific maintainer. This information can be used to see how well their packages are being maintained. If a maintainer has release critical bugs which have been outstanding for a while, this is a good indication that they are not performing their duties. This is especially the case if no activity can be found in the bug history, which is the case, for example, if the maintainer has never responded to the bug submitter asking for more information or clarifying the bug.

**FTBFS Bugs:** Debian supports a large number of architectures, and each package which is uploaded to the Debian archive is built automatically on all architectures. If a package has previously built on an architecture but no longer builds, a release critical bug is filed saying that the package "fails to build from source" (FTBFS). As with other release critical bugs, these give a good indication of the activity of a Debian maintainer.

In Debian's Bug Tracking system, bugs can be marked as closed in two ways. If the maintainer of a Debian package closes the bug, it is marked as done and the bug submitter is informed. On the other hand, if someone else closes a bug, it is merely marked as fixed and the bug submitter is not notified. It is the responsibility of the maintainer to confirm that the bug has been dealt with and to close the bug for good. This distinction is very helpful in the search for inactive maintainers: a large number of bugs marked as fixed might indicate that someone else is performing the maintainer's duties, maybe because they are not doing it themselves. Fixed bugs are especially interesting in the case of release critical and FTBFS bugs.

**Old Standards-Version:** The Debian Policy manual describes what Debian compliant packages have to look like, and this document is continuously updated to reflect new procedures. Each Debian package has a field which indicates which version of the Policy it complies with. This field, known as the Standards-Version, is very helpful to find packages which are out of date with regards to current Debian procedures. Similarly, if a package has not been updated a year after a stable release of Debian has been made, this is a good sign that a package is not maintained well. Even if a piece of software is not developed anymore, the Debian package has to be updated regularly as Debian procedures change. Hence, packages have to be updated regularly regardless of how fast the upstream software is developed (or whether it is still in development at all).

**New upstream version:** It is a good sign of an inactive maintainer if a new upstream version of the software has been available for a while which has not yet been packaged for Debian. While there are sometimes good reasons not to package the new version immediately (such as when a release is not deemed to be stable or of release quality yet), in most cases it hints at an inactive or busy maintainer.

### 4.1.3 Hints From Developers

With the current size of Debian, it is impossible for the fairly small quality assurance team to monitor each package to find inactive maintainers. It is therefore very beneficial to have a well-documented contact address where other developers and users can report maintainers who they think are not active anymore. The quality assurance team which has experience in tracking down

inactive maintainers can then use this pointer to investigate further whether a maintainer is really neglecting their duties.

## 5 Contacting Maintainers

Once a maintainer has been identified who is believed to be neglecting their duties, they have to be contacted before any measure is taken. There might be a good reason why they are momentarily not active and it is recommended to discuss the situation with them before anything else is done.

When contacting a maintainer who is believed to be inactive, it is important to be polite. After all, due to the distributed nature of free software projects, one may not be aware what happened to a volunteer. One always has to keep in mind that there might be a good reason why they are not performing their duties. For example, something might have happened to a relative or close friend. What they need the least at this point is a hostile e-mail asking why they are not spending time on their volunteer activities. It is even possible that something has happened to the volunteer and that a relative will read their e-mail.

Additionally, when contacting a maintainer, it is important to keep in mind that we are all volunteers. While I firmly believe that a participant in a free software project has certain responsibilities once they have volunteered, it is not consistent with voluntary work to expect someone to respond to your inquiry within a day. Similarly, one cannot demand from volunteers that they perform their duties. All you can do is politely ask that they do so, or ask them to officially let go so other volunteers can take over their tasks.

With this in mind, it is time to contact the maintainer. A short message summarizing one's findings, for example stating which packages are unmaintained, is in order. In the e-mail, one can politely ask what their situation is, and whether they still have the time and interest to carry out their duties. If the maintainer does not respond within two or three weeks, another message can be sent. This should refer to the first message, note that nothing has since been done and suggest that the packages should be given away so other maintainers can take over. In this mail, it should also be mentioned that action will be taken if the maintainer does not respond or does something soon. Again, if the maintainer does not react after two or three weeks, it can be assumed that the maintainer is really not active anymore. In this case, their packages should be given away so other maintainers can take over. In Debian, there is a listing for this known as "Work-Needing and Prospective Packages" [8]. Through this system, Debian developers can indicate to other maintainers that they are no longer interested in a particular package and that they are looking for a new maintainer.

At this point, it is also important to think about authority in the project. In Debian, there is not a written document which explicitly describes the authority of the quality assurance team. From this point of view, it is not clear whether they possess the authority to take packages away from a maintainer and make them available to others. However, this is a very important task which has to be done in order to maintain the quality of the whole system.

Hence, the quality assurance team started looking for inactive maintainers and took their packages away. The task was performed very carefully so no active maintainer was mistaken as inactive. Over time, other members of the project acknowledge that the quality assurance team had the authority to perform this function. While the authority was firmly established over time in the case of Debian, it is important to discuss this matter before taking action.

## 6 Tracking Inactive Maintainers

In large projects, such as Debian, there are many volunteers and it is possible that a large number are inactive. All of them have to be contacted and records have to be kept of who was contacted at what time and what the current status is. At some point, the number of people who have been contacted reaches a number where it is no longer feasible to keep the information in one's head. Therefore, a system is needed which allows one to easily keep track of what has happened so far. Such a system additionally allows multiple people to work on this task and to coordinate their efforts.

To make this possible for Debian, I have implemented a simple system which aims to be an effective means of keeping track of developer activity. New information can be added by e-mail, and the status of a maintainer or package can be queried on the command line. This collection of utilities, collectively known as `mia`, consists of three core tools:

**mia-record:** This tool processes e-mail and stores the information. When an inactive maintainer is contacted, a blind copy of the message can be sent to a mail alias which pipes the message into `mia-record`. The tool will store the e-mail and ask for a summary of this message by e-mail. Once the e-mail is answered, a summary of the message is stored. Alternatively, an `X-MIA-Summary` header can be added to the original message and that summary is stored automatically.

**mia-history:** This tool shows the previously added summaries of a specific volunteer or of all volunteers who have been recorded so far. This tool is hence very useful to easily see what has happened already. For this tool to be helpful, it is crucial that the summaries supplied to `mia-record` are content-rich. Additionally,

it is useful if the summaries correspond to a specific schema so they can be easily parsed automatically. For Debian, I have introduced a set of arbitrary keywords which summarize different states. As an example, the output of `mia-history` might look like this:

```
foo:
    2002-01-06: Hint: bar
    2002-04-02: lost interest
    2002-04-03: Orphaned: bar
```

The person `foo` was contacted because someone gave a hint that the package `bar` was in a bad shape. The maintained responded saying that they are no longer interested in maintaining the package, and therefore the package was given away ("orphaned" in Debian's terminology).

**mia-check:** If proper keywords are used in the summaries, `mia-check` can automatically show which volunteer has any tasks outstanding. For this to work, different keywords belong to certain classes. The keywords "hint" adds an item to the TODO list of a volunteer. There are other keywords, such as "RC" (release critical bugs) or "S-V" (the Standards-Version is very old, i.e. the package does not conform to current policies), which add items. On the other hand, keywords such as "orphaned" or "removed" take an item from the TODO list. `mia-check` goes through all keywords, and if anything is left on the TODO list at the end and the volunteer has not been contacted recently, `mia-check` shows that this volunteer has to be contacted again.

While these tools are very simple, they make the work tremendously easier. They allow the quality assurance team to keep track of what has been done already. Data is stored with `mia-record`, queried with `mia-history`, and `mia-check` is a useful reminder showing outstanding tasks. Once a maintainer is contacted several times without response, it is time to act and make the packages available to other developers.

While these tools allow easy track keeping, I firmly believe that finding and contacting inactive maintainers cannot be fully automated in a sensible way. Some tasks, such as obtaining an overview of the packages of a maintainer, can be automated or at least semi-automated, but the whole process of identifying and dealing with inactive maintainers relies on human judgement. This makes the whole process very time consuming. However, in order to maintain the quality in a system inactive volunteers who play important roles have to be identified and solutions found.

# 7 Debian and Inactive Maintainers

Debian has recognized the problem of maintainers who neglect their packages and began approaching the issue systemically in 2001. Since then, various activities have been carried out. In particular, two different kind of activities can be distinguished. First, there is a continuous effort to track down inactive maintainers and to sort out solutions for their packages. Second, all maintainers without packages were contacted once to see whether they are still interested in volunteering for Debian.

## 7.1 Continuous Activities

Since 2001, there have been continuous activities to find and contact inactive package maintainers, and in case they do not respond to make their packages available to other developers or remove them if they are obsolete. From 2001 to the end of 2003, around 180 package maintainers were contacted, more than 320 packages were given away to other maintainers and around 25 packages were removed. These figures suggest that finding and dealing with inactive maintainers might have a real impact on quality.

## 7.2 The MIA Ping

In addition to the continuous activities carried out by the quality assurance team, the Debian Account Manager, who has the authority to add and remove volunteers from the project, conducted a 'MIA ping'. He contacted every Debian developer without a package in the archive in March 2003 to determine whether they are still interested in volunteering for Debian. If they did not respond to the mail after two months, their account would be deactivated and they would be put in an "emeritus" class. Recent compromises of GNU [10] and Debian [6], both involving local root exploits, show the importance of purging old users from project machines.

This MIA ping revealed that the e-mail of 34 volunteers was bouncing, 94 volunteers did not respond at all, 28 wanted to retire officially, 10 said they were still active, and 26 expressed their continued interest but it was not clear whether they would really do any work. It turned out later that the majority of those who were not sure about their involvement remained inactive. These numbers are very interesting because it was the first time in Debian's ten year history that anything like this was carried out. It helps to get a better grasp of the actual number of volunteers in a project, and improves security as unused accounts can be removed or locked.

# 8 Proposed Remedies

There are many factors which contribute to the severity of the problem of inactive volunteers. In addition to the distributed and volunteer nature of free software projects, tracking inactive volunteers is a complicated task because it cannot be fully automated. It requires human judgement to distinguish whether a volunteer is temporarily busy but likely to return or whether they are really inactive and neglecting their duties. Some

mechanisms for finding inactive volunteers can be automatated, but the whole tasks remains very time intensive. Furthermore, there is usually a long time span between realizing that a volunteer is neglecting their duties and something is done about it. As I argued before, it is polite so send several messages and to wait for two or three weeks for a reply to each message. Therefore, it can take months until the problem is resolved and the situation improves. During that time, the quality of the software suffers.

In order to keep quality high at all times, it is vital to take preventive measures. For example, the admission process should take the problem of inactive volunteers into account and make sure that prospective volunteers understand the problem. Another recommendation is to limit the introduction of low-interest packages into the distribution. A package that only one developer is even potentially interested in maintaining carries a large risk of being neglected or abandoned. Another possible way to assure quality is to have a dedicated group of developers who can temporarily maintain a package when its maintainer is busy. Finally, another recommendation for Debian is to move away from having a single maintainer per package to having teams who are responsible for a package [15]. This way, more redundancy is created and the reliance on a specific volunteer is smaller.

Fortunately, an increasing number of volunteer projects, including GNOME [21] and KDE [13], are paying more attention to quality assurance, and it is my hope that mature QA processes for free software projects will be developed in the next few years. It would also be beneficial if free software hosting sites, such as SourceForge, would actively classify inactive projects as such and had quality assurance teams.

## 9 Conclusions

Volunteers who neglect their duties are a potential problem in any free software project. In large projects, it is often difficult to recognize immediately when a volunteer does not carry out the tasks they are responsible for. Unlike in commercial companies where it is obvious when an employee does not come in to work, it requires much effort in distributed, volunteer projects to find who is inactive. While there are various sources of information which can be taken into account, the process of finding inactive maintainers and contacting them is time intensive and requires human judgement. It is therefore impossible to fully automate the task. The process of moving from realizing that a volunteer might be inactive to resolving the situation also takes a long time as it is polite to first contact the volunteer to see whether they can explain the situation. During this time, quality in the project decreases as the functions the volunteer is responsible for are not carried out. Due to this, it is important to take preventive measures and to consider this problem up front.

In this paper, I have described Debian's approach to the problem of package maintainers who do not perform their duties. Several sources of information have been introduced which are used in tracking down inactive maintainers. While these sources of information are specific to Debian, they help other projects gain a better understanding of the problem and this allows them to develop strategies which apply to their respective projects. Various tools have been described which are used to keep track of inactive maintainers in Debian and which allow multiple members of the quality assurance team to work on the problem together.

Free software projects have to acknowledge their volunteer nature and introduce more redundancy. They have to realize that certain volunteers will become inactive at some point, and take precautions. The problem of volunteers neglecting their duties has to be recognized and dealt with in order to maintain the high quality found in many free software projects as well as an effective development process.

## 10 Availability

The tools mentioned in this paper which are used in Debian to track inactive maintainers are available under the GNU General Public License (GPL) from `http://cvs.debian.org/mia/?cvsroot=qa`. Debian consists completely of free software as per the Debian Free Software Guidelines and is available from `http://www.debian.org/`.

## 11 Acknowledgements

This work was in part funded by the NUUG Foundation. I would like to thank Bart Massey for his valuable comments and suggestions.

## References

[1] Debian Bug Tracking System. `http://bugs.debian.org/`, accessed April 6, 2004.

[2] Debian constitution. `http://www.debian.org/devel/constitution`, accessed April 6, 2004.

[3] Debian Developer's Reference. `http://www.debian.org/doc/developers-reference/`, accessed April 6, 2004.

[4] Debian New Maintainer process. `http://www.debian.org/devel/join/newmaint`, accessed April 6, 2004.

[5] Debian Policy. `http://www.debian.org/doc/debian-policy/`, accessed April 6, 2004.

[6] Debian Project machines compromised. `http://www.debian.org/News/2003/20031121`, accessed April 6, 2004.

[7] Debian Social Contract. `http://www.debian.org/social_contract`, accessed April 6, 2004.

[8] Debian Work-Needing and Prospective Packages. `http://www.debian.org/devel/wnpp/`, accessed April 6, 2004.

[9] Michael E. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3), 1976.

[10] GNU Project FTP server compromised. `http://lwn.net/Articles/44310/`, accessed April 6, 2004.

[11] Jesús M. González-Barahona, Miguel A. Ortuño Pérez, Pedro de las Heras Quirós, José Centeno González, and Vicente Matellán Olivera. Counting Potatoes: The Size of Debian 2.2. *Upgrade*, II(6):60–66, December 2001.

[12] Guido Hertel, Sven Niedner, and Stefanie Herrmann. Motivation of software developers in open source projects: an Internet-based survey of contributors to the Linux kernel. *Research Policy*, 32(7):1159–1177, 2003.

[13] KDE Quality Team. `http://quality.kde.org/`, accessed April 6, 2004.

[14] Stefan Koch and Georg Schneider. Effort, cooperation and coordination in an open source software project: GNOME. *Information Systems Journal*, 12(1):27–42, 2002.

[15] Martin Michlmayr and Benjamin Mako Hill. Quality and the reliance on individuals in free software projects. In *3rd Workshop on Open Source Software Engineering*, pages 105–109. ICSE, 2003.

[16] Audris Mockus, Roy T. Fielding, and James D. Herbsleb. Two case studies of open source software development: Apache and Mozilla. *ACM Transactions on Software Engineering and Methodology*, 11(3):309–346, 2002.

[17] Glyn Moody. The greatest OS that (n)ever was. *Wired*, 4 August 1997.

[18] Eric S. Raymond. *The Cathedral and the Bazaar*. O'Reilly & Associates, Sebastopol, CA, 1999.

[19] Eric S. Raymond. *The Art Of Unix Programming*. Addison-Wesley, 2003.

[20] Stephen R. Schach, Bo Jin, David R. Wright, Gillian Z. Heller, and A. Jefferson Offutt. Maintainability of the Linux kernel. *IEE Proceedings - Software*, 149(1):18–23, 2002.

[21] Luis Villa. Large free software projects and Bugzilla. In *Proceedings of the Linux Symposium*, Ottawa, Canada, 2003.

# Creating a Portable Programming Language Using Open Source Software

Andreas Bauer
*Institut für Informatik*
*Technische Universität München*
*D-85748 Garching b. München, Germany*
baueran@in.tum.de

## Abstract

On a first glance, the field of compiler construction and programming language design may not seem to have experienced major innovations over the last decade. By now, it is almost common knowledge how a lexer works, how parsing is done, but not many have yet realized how Open Source software — and in particular the GNU Compiler Collection — have silently offered language implementors new and better ways to do their job. Therefore, this paper describes the novel advantages Open Source software provides and, furthermore, it illustrates these with practical examples showing how the presented concepts can be put into practice. Another important contribution of this paper is to give an overview over the existing limitations and the technical problems that can occur when creating an Open Source based programming language implementation.

## 1 Introduction

The extensive Open Source GNU Compiler Collection (GCC) offers optimized code generation for a large number of different platforms and programming languages, for instance, C, C++, Java, and Ada, to name just a few. Historically, however, GCC was like most other compilers, aimed to support only one programming language, namely C, and for only a limited number of target platforms, i. e. those that would support the GNU system [1].

Due to the openness and the free availability of the source code, GCC was soon retargeted to, back then, even exotic hardware platforms and the differentiation between the "old" GNU-C back end, and the separate front ends became increasingly immanent. In other words, GCC turned into a quasi-platform itself, rather than being just another C compiler.

For programmers implementing a new language, this development can be of tremendous benefit, because it means that they can rely on GCC as being their actual target, so that native code generation is basically transparent to the front end. In a nutshell, it allows the implementors to focus on what is really important for them: language design.

Fig. 1 shows the different compilation stages of the compiler suite: the input can either be a straightforward C program code or, alternatively and more interestingly, interfacing occurs where the dashed line separates the stages. This leaves users with the choice of *a)* interfacing GCC in an "old fashioned" manner by emitting standard C code as well, or *b)* by interfacing the back end directly via the *tree structure*. This data structure is GCC's means to describe abstract syntax trees. Technically, however, a `tree` is merely a GCC-specific pointer type, but the object to which it points may be of a variety of different types; it is used to represent and perform various optimizations on the program (see § 4.3, 5).

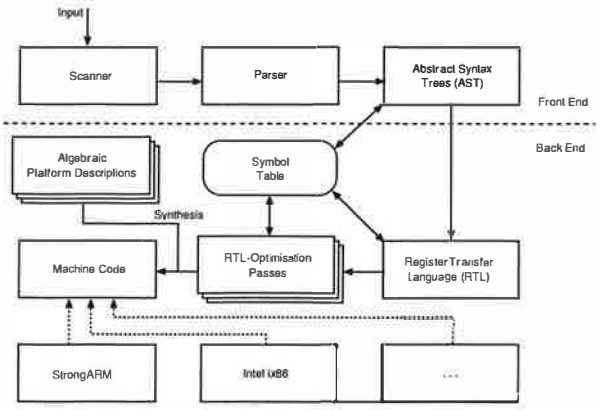



*Figure 1: The main compilation stages of the GCC suite.*

Although many compilers do indeed translate a program merely to C code, they potentially sacrifice the possibility of generating useful as well as detailed debugging information regarding the input program, and are also likely to miss out on specific intermediate program optimizations performed on the tree structure, e.g. alias analysis. Hence, the focus of this paper rests solely on integrating a well defined GCC front end that employs trees to communicate with the compiler suite's back end.

For the sake of completeness, it should be pointed out though that targeting (low, or high level) GNU-C code does offer advantages compared to, say, emitting

ANSI-compatible C. GNU-C extends ANSI-C with various non-portable constructs that help emitting optimized machine code. Nested functions, or global register variables are just some of the many characteristics custom to GNU-C [1].

An exceptional thing to note about Fig. 1 is how GCC handles the generation of native binary code: by consulting a separate, more or less, algebraic specification of the actual platform, such as `i386-gnu-linux-aout`, the abstract machine code written in the Register Transfer Language (RTL) gets mapped to native machine code according to the physical reality of the respective targets. This "physical reality" is typically determined by the available set of commands, number of processor registers, employed calling conventions, and so forth.

Such a strict differentiation between the various processes and the strong modularization of the GCC suite as it is also reflected in the figure, essentially, makes many of the typical tasks a compiler writer has to go through [2] redundant. For instance, complicated basic block analysis, or register coloring algorithms in the back end do not need to be re-implemented any longer. Instead, by using the compiler suite, all the effort can be put into designing the distinctive and essential features of a new programming language. Back end optimizations are already in place.

Problems, of course, remain. Although the GCC front ends do not need to be concerned about hardware constraints in the first place, mapping from higher level language features into GCC's interfacing `tree` representation can be all, but trivial. As a matter of fact, certain types of programming language front ends have been battling with GCC's code generation strategies for quite some time. (See, for instance, [3].) However, this is not necessarily so, because the program representation in terms of a tree structure is inadequate, but rather due to the fact that GCC treats that intermediate program representation, basically, as if it was a procedural (C-like) program. In a lot of cases this is not a problem, in others, however, very subtle problems arise.

Consequently, this paper not only goes through the notion of targeting GCC as a portable back end (see § 4, 5), but it also hints to practical problems that may arise when applying the presented concepts in a straightforward naïve manner (see § 6).

## 2 Related Work

Especially in light of commercial software development, GCC is by far not the only portable back end solution, even though it is probably one of the most accessible and useable ones today, due to the free availability of the sources and the active community surrounding it.

### 2.1 Back Ends

For instance, Chess/Checkers [4] is a very successful, commercial framework that is suited particularly well to build embedded systems software. The Chess module acts as a — more or less — standard C compiler while the subsequent passes of Checkers map the output to a user specified architecture written in a specialized hardware description language.

But also in the Open Source world, further portable back ends do exist, e. g. MLRISC [5] which is a customizable optimizing back end written in Standard ML that has also been successfully retargeted to multiple (mostly RISC) architectures. It deals with the special requirements imposed by the execution model of different high level, typed languages by allowing many components of the system to be customized to fit the source language semantics as well as the runtime system requirements.

A framework which comes close to GCC's ideals is the Little C Compiler (lcc) [6]. Basically, it is a retargetable compiler for standard C and generates native code for the Alpha, Sparc, MIPS R3000, and Intel i*x*86 as well as for its successors. Directly interfacing with lcc is different to GCC though and, therefore, not discussed in this paper.

Additionally, new programming languages could also use (parts of) the free Zephyr environment [7] which also offers concepts that are similar to those found in the GCC suite: for instance, the Zephyr component VPO is a platform and language independent optimizer that is built upon its own register transfer language. It has already been used with several C front ends, each of which generates a different intermediate language.

Obviously, the choices are manifold, but for the remainder of this paper the focus rests solely on the GCC back end, simply because it is the most widely used of all the presented suites, has a very large and active developer community, and is open and free in the sense of the GPL to allow and, in fact, encourage modifications to it.

### 2.2 Front Ends

Already a large number of free (and not so free) programming languages make use of a separate, portable back end. The increasingly popular logic language compiler Mercury [8], for instance, offers even multiple interfaces; among them is one for "pure" low level C, GCC trees, Java, and lately even .NET support was added.

The Glasgow Haskell Compiler (GHC) [9] is another prominent compiler for a declarative language, namely Haskell, that achieves portability thanks to the GCC back end. Although GHC offers its own optimizing code generation for Sparc and Intel i*x*86 processors, it still relies on the GCC suite when a wider range of hardware

targets needs to be addressed. GHC, on the other hand, is also a perfect example where straightforward interfacing fails. The reasons for that are outlined in greater detail in § 6.

Of course, further interesting language implementations based on GCC exist. Aside from C, the standard distribution already supports Java (GJC), Ada (GNAT), Fortran (G77), Treelang, Objective-C, and C++ (G++). However, when considering the integration of a new language these may not turn out to be the best starting points, since each such front end is rather sophisticated in itself, and the essential interface mechanisms to GCC cannot be seen clearly. This paper aims to narrow this documentation gap at least partly.

## 3 A Toy Expression Language

This section introduces the foundations of a rather simple expression language which will be used as an example throughout the remainder of this text. Let's call the language *toy*, simply because it does almost nothing useful. Toy is inspired by the "pocket calculator language" hoc as it is described in [10] and, similarly, in other text books.

*list* ::= **empty**
| *list*
| *list fnbody*

*assignment* ::= **variable** = *expr*

*fndecl* ::= **name** ( )
*fnbody* ::= *fndecl* : **begin** *expr* **end**

*expr* ::= **number**
| **variable**
| *assignment*
| *expr* + *expr*
| *expr* − *expr*
| *expr* * *expr*
| *expr* / *expr*
| ( *expr* )
| −*expr*

*Figure 2: The abstract syntax for the sample language, toy.*

An abstract syntax for our sample language can be seen in Fig. 2. The bold font is used to reference to tokens which are handled separately. Ambiguities in the grammar (usually resulting in shift-reduce conflicts) can later be resolved by assigning the respective yacc precedences for each operator. Note, for the sake of simplicity, loop structures and the like have been omitted from the grammar. However, the sections § 4.3–5 address the processes of including additional as well as more advanced language features.

Basically, toy covers the most elementary operations of arithmetics and allows the user to hold temporary values in variables. Hence, a toy implementation accepts only very basic programs and does not issue any warnings at compile time; that is, a toy program is either correct, or incorrect.

## 4 Interfacing directly with GCC

Despite a number of already existing front ends for GCC, interfacing its optimizing back end is a process that can be all, but easy to conceive. This is mainly due to the fact that *a)* the interface itself does not remain 100% stable and has evolved in an ad-hoc manner along with each additional front end, and *b)* the entire process as well as the interface itself are not properly documented anywhere [11] (especially if one considers file dependencies and the like as part of the actual interface).

### 4.1 Essential Files

Most importantly, when writing a new compiler based on GCC the user has to provide a number of essential files such as `Make-lang.in`, or `config-lang.in` which describe the build dependencies as well as the name of the new language, unique suffixes (`.toy`), and, finally, the name of the executable compiler (`cc1toy`). These files are all located in a separate directory inside GCC's main source directory, e. g. `gcc-3.3.2/gcc/toy`.

Thanks to the rather accurately documented Makefiles of already shipped front ends, it is almost obvious how a rudimentary `Make-lang.in` should look like: GCC merely expects a number of build, install, and clean targets in order to integrate the new front end into the overall compilation process. In other words, the shell command '`make toy.clean`' should be functional to clean up merely the toy object files, while '`make toy.dvi`' should produce a printable manual in dvi-format, and so on.

Note, that the executable `gcc` is usually built as the main *compiler driver*, i.e. it recognizes and "drives" the user supplied source code to the according compiler, based on input file suffixes and command line options.

### 4.2 Methods of Interfacing

When creating a programming language, the user typically has a couple of different choices on how GCC can be integrated to achieve a maximum of portability later on. The most obvious and probably most widely used approach is to create a custom front end which is subsequently tied to the compiler suite's driver. However,

depending on the complexity and size of the new language implementation it is also possible and feasible to link merely the GCC back end to a stand-alone front end whose back end driver would then be a shared library. One language that follows this approach is Mercury [8]; here, the difference for users is mainly noticeable when invoking the compiler as the compile driver is not the `gcc` executable, as usual, but rather the new front end itself.

Either method affects primarily the configuration nitty gritty of the new compiler, because in principle a user can choose whatever language seems suitable for implementation as long as there exists at least a certain degree of compatibility with the C-language's calling convention (or, at least, some sort of an interface to it), essential for linking the parts together in the end.

Additionally, there is also a choice on whether to use GCC's internal data structures for code representation directly, or not. In other words, the front end could build a valid GCC tree structure, e.g. in the most simple form by using semantic actions in the attributed grammar, or it could build up and fill its own intermediate data structures which are subsequently translated into GCC trees and, finally, RTL.

Of course, both approaches are valid, and having to create a custom intermediate code representation in the front end is very tedious work. The advantage of going through the effort, however, is a better possibility to trace bugs in the user submitted code, because the front end would then be able to check additional constraints that are normally hard to tackle in later stages of the compilation process inside the GCC back end.

Also, when making up a GCC-based programming language entirely from scratch users are likely to face difficulties in situations where they have to interface directly with the `tree` structure: once the back end is given a malformed `tree`, it is hard to undo or rebuild parts of it, let alone associating the error with the originally provided piece of code. Therefore, the creation of an intermediate program representation first can be — despite a reasonable amount of extra work – sometimes a good idea if the supported language is rather challenging in terms of representing its core features, e.g. special scoping, nested inner functions and classes, etc.

## 4.3 The `tree` Representation

The `tree` structure acts as the main interface between a custom front end and GCC's back end. A good overview over all the pre-defined trees is available, for instance, in [1] but details are really only documented in the GCC source files `gcc/tree.def` and `gcc/tree.h`. Consulting and understanding these files is absolutely essential for our task.

A `tree` is really only a pointer type representing a single node of an abstract syntax tree structure. It can be used for a *data type*, a *variable*, an *expression*, or a *statement*. Each node has a "tree code" associated with it which says what kind of object it represents. For instance, the code

- `INTEGER_TYPE` represents a type of integer,
- `ARRAY_TYPE` represents a type of pointer,
- `VAR_DECL` represents a declared variable,
- `INTEGER_CST` represents a constant integer value,
- and `PLUS_EXPR` represents a plus-expression (see Fig. 2).

The structure to which the pointer points to is implemented via a C union type. The individual fields are accessed via *predicates*, i.e. C-type macros, such as `INTEGRAL_TYPE_P` which in turn calls and evaluates the result of `TREE_CODE` to determine whether a given node is of integral type. Although in general, it is feasible to apply any type of tree node to a predicate, there are certain macros that demand for nodes of exactly a certain kind; `TREE_CODE`, however, is not one of them. Common fields are summarized in the structure `tree_common`.

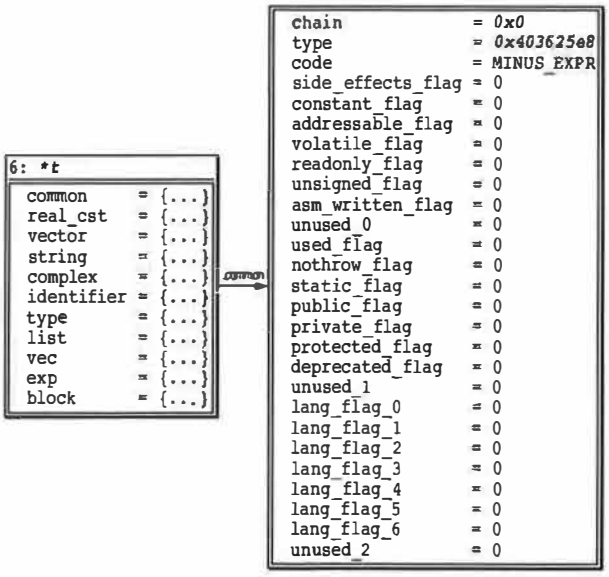



*Figure 3: A binary minus expression node, visualized using the Data Display Debugger. The `tree_common` structure is right of the `tree` union definition.*

The inside of an expression node is depicted in Fig. 3. Its operands can be accessed with the macro `TREE_OPERAND` as in `TREE_OPERAND(my_expr, 0)`; that is, an expression's operands are actually zero-indexed.

When writing a properly integrated GCC front end, i.e. one that issues trees, it is most useful to take

text.c:

```
int
main ()
{
  int result_0;
  compute (5);
  return result_0;
}

int
compute (int argument_0)
{
  return argument_0 * 2;
}
```

text.c.tu:

```
@1      type_decl          name: @2       type: @3       scpe: @4
                           srcp: <internal>:0            chan: @5
@2      identifier_node    strg: int      lngt: 3
@3      integer_type       name: @1       size: @6       algn: 32
                           prec: 32       min : @7       max : @8
@4      translation_unit_decl srcp: <internal>:0
        :
@1969   identifier_node    strg: main     lngt: 4
@1970   function_type       unql: @1787    size: @20      algn: 64
                           retn: @3
@1971   function_decl       name: @1972    type: @1453    scpe: @4
                           srcp: test.c:9                args: @1973
                           extern
@1972   identifier_node    strg: compute  lngt: 7
@1973   parm_decl           name: @1974    type: @3       scpe: @1971
                           srcp: test.c:8                argt: @3
                           size: @6       algn: 32       used: 1
@1974   identifier_node    strg: argument_0              lngt:
```

*Figure 4: On the right hand side is a typical dump for a single translation unit generated by the C front end of GCC. Boxed are the distinctive function definitions and arguments as they appear in the original source code on the left. The @-symbols identify and reference tree nodes as is also indicated by the arrows.*

a closer look at the generated "tree code". The C and C++ front ends allow for this already using the `-fdump-translation-unit` switch, amongst others. A typical result of this can be seen in Fig. 4 and, although extensive and complex, the typical properties described above are all present in this short example. Furthermore, the additional "links" drawn in the figure give a rough idea about how well suited the tree structure is to extract control flow information from a translation unit.

## 5 Mapping

In order to map a toy statement covered by a rule $expr ::= expr + expr$ to a GCC binary expression node, the pre-defined expression `PLUS_EXPR` can be used, or accordingly, `MINUS_EXPR`, and `MULT_EXPR` for the other basic arithmetic operations. (Division is being treated separately due to data type and rounding issues.) Of course, additional expressions exist to hold further data types, single programming language statements, or even entire function definitions as well as general compound statements.

### 5.1 Generating Trees

For many languages, most of the tree generation occurs during parsing. Consequently, such front ends make use of a yacc-like grammar (see Fig. 2) to be able to employ other Open Source tools like GNU Bison, for instance. Bison uses an LALR(1) algorithm (look-ahead LR) to recognize a context-free language. Some front ends, like the C++ of GCC >= 3.4, however, use and have introduced their own *recursive descent* parsers.

There is no single "best way" of parsing, but it is noteworthy that there exist many programming languages for which a context-free grammar in general is not expressive enough. Again, the purely functional language Haskell provides an example for being one of them [12, § 9.3]. For a language like toy, however, the tools yacc, or GNU Bison are fully adequate. A good practice would be to hook them into `Make-lang.in` and to dynamically (re-)build the parser along with the actual front end. This also applies to lexicographic analysis via flex; flex, the "fast lexer", is usually a very good companion for this exercise, but not scope of this paper. (See [10], or `flex(1)` for further details on flex and yacc.)

To assemble a valid tree structure for a toy expression language, yacc *actions* can be used. (Other parsers have to invent their own.) That is, the implementation of the grammar shown in Fig. 2 gets extended by statements like

```
expr:  number        { ... }
     | variable      { ... }
     | expr '+' expr
       { $$ = build
              (PLUS_EXPR, /* Tree code */
               integer_type_node,
               $1,        /* Operand 1 */
               $3);       /* Operand 2 */ }
     | ...
     ;
```

where `build` is a pre-defined function of `src/tree.c`. It gets used to build a binary expression of a certain tree code with a certain type.

It is important to notice that the largest possible tree structure is always built on a per-unit basis which usually resembles an entire input function. Hence, the parser

typically interrupts at *a)* each function declaration, *b)* each definition, and *c)* sometimes also at each explicit function closing. Since toy functions are defined to be argument-less, the according actions could be implemented as follows:

```
fndecl:  name '(' ')'
         { $$ = build_fndecl ( $1 ); }
fnbody:  fndecl ':' BEGIN expr END
         { build_fnbody ( $1, $4 ); }
```

Basically, `build_fndecl` needs to call the according functions of `src/tree.c`, firstly to create a node representing the parsed function's return as well as argument types (`build_function_type`), and secondly to hold the declaration itself (`build_decl`). `build_fnbody` needs to contain code for building tree nodes that represent the return value as well as for emitting actual RTL. Both routines are schematically depicted in Fig. 5.

Historically, the transformations had to be performed statement by statement in order to facilitate RTL synthesis and to avoid space problems, but this narrow scope turned out too constraining for complex programs. Nowadays, GCC enjoys internal *garbage collection* (GC) to tackle the memory issues, which is almost always recommended to be used in front ends, too (see § 6.3).

## 5.2 From Trees to RTL

Basically, the `tree` data structure acts as the main interface between a GCC front end and its optimizing back end. However, a front end is not totally isolated from the technicalities of emitting RTL. Typically, it has to

- ensure that there exists a transformation from all the employed tree types to RTL;
- trigger RTL expansion at the end of parsing functions, or general compound statements;
- provide direct tree-to-RTL conversion, should custom tree types be involved that cannot be lowered to existing ones (see also § 5.3).

Although, `build_fndecl` does not directly emit RTL, the call to `src/tree-optimize.c:tree_rest_of_decl_compilation` does, in fact, trigger subsequent low level code generation. The function itself is a flexible wrapper around `src/toplev.c:rest_of_decl_compilation` and can also handle features like nested input functions should it be required.

In `build_fnbody`, the user has to deal with RTL expansion more directly. As a rule of thumb, `expand_*` functions usually accept a `*_STMT`, or `*_DECL` node and emit RTL for it thus, `src/function.c:expand_function_start` starts RTL for a new function and must always be called first.

Omitted are the details of handling arguments and garbage collection, since toy functions are rather basic with only one type, integer, and zero parameters each. Additionally, these aspects are very front end specific and can be added afterwards, once a basic front end compiles.

In principle, the internal garbage collector is accessed via different `ggc_*` calls:

**ggc_add_tree_root:** This is used to hook into the marking algorithm of GCC and should come first.

**ggc_alloc:** In order allocate memory, `ggc_alloc` has to be used.

**ggc_collect:** This function triggers the top-level mark-and-sweep routine. It gets called several times by `src/toplev.c:rest_of_*` functions to free memory.

The file `src/toplev.c` also provides the main entry point for the C and C++ front ends, i.e. a `main` routine, which in turn invokes the various compilation passes. Obviously, the file also handles a lot of the code generation from trees and, therefore, can and should be used directly wherever possible. Reimplementing `src/toplev.c` stuff requires a very thorough understanding of the GCC internals and is, definitely, one of the most difficult parts when building a compiler from scratch.

## 5.3 Introduction of new Tree Types

The pre-defined data types, tree codes and macros as they are available in GCC versions 3.$x$ are — for a basic expression language as it is described in § 3 — sufficiently expressive to be able to create a comprehensive intermediate code representation which then gets mapped to RTL instructions.

The downside, however, is that currently GCC front ends behave somewhat like a C compiler, in a sense that the syntax of the tree structure is strongly biased towards procedural languages. In other words, the front end of a language more sophisticated than toy, or C, probably based on an alien programming paradigm, almost always needs to introduce its own tree definitions to adequately represent statements, functions, and all kinds of additional types (see § 6, 7).

Introducing new kinds of trees happens in the respective `.def` files of each front end, e.g. simply via `DEFTREECODE (PLUS_EXPR, "plus_expr", '2', 2)`: the first operand is the tree *code*, the second is its *type*, the third is its "kind" (i.e. used for constants, declarations, references, binary arithmetic expressions, and so on), and the optional fourth argument is the number of argument slots to allocate if necessary; two in this example.

```
tree
build_fndecl (char* name)
{
  ...
  my_fntype = build_function_type
              (integer_type_node,
               param_type_list);
  ...
  my_fndecl = build_decl
              (FUNCTION_DECL,
               name,
               my_fntype);
  ...
  tree_rest_of_decl_compilation (my_fndecl, 0);
  return my_fndecl;
}
```

```
void
build_fnbody (tree fndecl, tree expr)
{
  expand_function_start (fndecl, 0);
  ...
  expand_return (build
                 (MODIFY_EXPR,
                  void_type_node,
                  DECL_RESULT (fndecl),
                  expr));
  ...
  expand_function_end (...);
}
```

*Figure 5: These two functions are responsible for building trees representing an input function's declaration, as well as for holding its definition, type, and to trigger RTL generation for each node, respectively. Naturally, `build_fndecl` should be called first. Calls to `expand_*` denote direct RTL expansion.*

On the one hand side, the extendibility allows for a representation of almost arbitrary programming language features and types, but on the other it also requires additional work to *a)* either lower the extensions to the existing nodes, or *b)* to make up additional `expand_*` functions that match, say, an `*_STMT` node to RTL. The expansion functions mostly reside in `src/stmt.c`, making up custom ones, however, is more challenging than lowering and one of the reasons why RTL is not the interface of choice between the front and the back end of GCC.

## 6 Problems

Although, GCC offers marvelous possibilities to speed up the development of rather platform independent programming language implementations, it does put its own peculiar constraints on front ends, especially if those resemble non-imperative paradigms.

Most prominently, functional and logic programming languages have a hard time taking full advantage of a "generic" back end like GCC. Many of them offer quite advanced concepts such as first order and higher order functions, automatic garbage collection, and a strong static type system based on polymorphism — clearly a contrast to the C language.

### 6.1 Higher Order Functions

Many programming languages treat functions as first-class citizens to support higher order functions. A higher order function takes one or many functions as argument, or returns these as its "return value". This is especially useful when using the *continuation passing* style (CPS) with the continuation being a passed-on function that should be executed next, similar to an additional program counter. In other words, the order of the computation is implicitly defined by an additional function argument — the continuation. Note, real CPS functions never return.

CPS gets employed in many functional programming language compilers and interpreters, such as GHC used for Haskell, for instance. But, unlike in Haskell, functions are not first order in C nor in GCC, i.e. functions can not be passed as an argument. Therefore, higher order functions do not exist in C. Function pointers, however, are first-class and the continuation passing style can be approximated by using `void` pointers to functions.

```
-- a) Without a higher order function
squareListNoHof []   = []
squareListNoHof list = ((head list)^2):
                       (squareListNoHof
                        (tail list))

-- b) With a higher order function
squareList list = map (^2) list
```

*Figure 6: These two Haskell algorithms compute the square of each value given in a `list`, in* a) *without the use of higher order functions, and in* b) *in combination with the higher order function map which takes a function and a list as arguments. Clearly, solution* b) *is more compact, thus easier to comprehend.*

In essence, this means that a lot of explicit casts and indirect function calls are involved. Since function pointers can only refer to functions with either global scope, or local scope to a particular file, this approach itself is not suitable to get the full flexibility of higher order functions which users may be used to from their declarative programming language of choice (see Fig. 6). Higher order functions may also be lexically nested and need *closures* to represent such scope information (see § 6.3).

Consequently, declarative languages like Haskell, ML, or Lisp demand for a compiler which provides its

own mechanisms for handling higher order functions such that the GCC back end must not be overly concerned with that "technicality" anymore. For example, GHC tackles this problem using a twofold approach: firstly, it maintains its own internal stack structure parallel to the architecture's runtime stack, and secondly, by modifying the output of GCC using a crude pattern matching algorithm that removes certain assembly instructions that are responsible for handling function arguments and frames [13]. This, however, is also described in greater detail in § 6.2.

## 6.2 Tail Calls

In all declarative programming languages a high number of recursive function calls occur of which, typically, many are tail calls. A tail call is a function call in the tail position of the calling function.

Consider the following straightforward Haskell implementation of the greatest common divisor algorithm:

```
gcd :: Int → Int → Int
gcd a b | a == b = a
        | a > b  = gcd (a - b) b
        | a < b  = gcd a (b - a)
```

Typical for the declarative programming style, this code is relatively easy to conceive; it contains two tail recursive calls to calculate a result. In that vein, most functional code bears a high percentage of tail calls which are either recursive, or even mutually recursive with a differing number of function arguments, respectively.

Tail calls can be implemented without requiring more than a single stack frame of the architecture's runtime stack, regardless of the amount of tail calls performed. This is possible, because a tail call is, essentially, the very last instruction of the caller. Thus, the caller has finished computation at this point and its stack frame should be free for "recycling" by the callee [3, 14] which would then take over responsibility to return to the original caller, or issues further tail calls.

According to the UNIX calling convention for C, however, the caller is responsible for cleaning up the callee's function arguments (see [15]). Thus, marshalling arguments subsequently to the actual tail call has to be realized in such a way that the topmost caller either does not remove a wrong number of arguments, or lets the callee discard unused stack slots itself. Fig. 7 shows what could happen, for instance, when a function $f(A1, A2, A3)$ performs a tail call to $f'(A1, A2, A3, A4)$ which expects an additional `int` argument.

Although elegant, tail calls impose a number of problems to the user who seeks to employ GCC as a back end for, say, a purely functional programming language compiler such as as GHC:

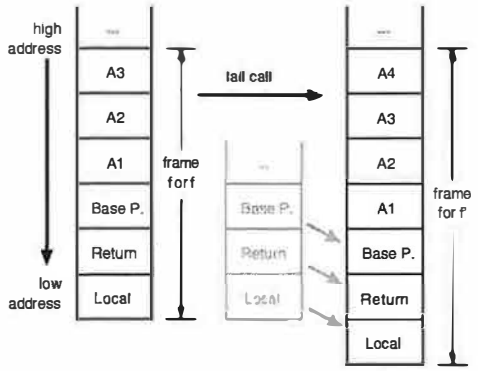



*Figure 7: A proper tail call to a function which takes more arguments than the caller requires runtime stack marshalling for reusing the current stack frame, e.g. shifting the return address down (or up, depending on the platform).*

1. Using recursion exclusively as a means of computation has not been foreseen by most platform's C calling convention [15]. Typically, the calling conventions do not differentiate between a tail call and a "normal" call. Hence, a stack frame gets always reserved via the `call` command of the according architecture and leads to a stack growth rate of $O(n)$ — linear but malicious.
2. Because of 1. proper tail calls with an overall stack consumption of $O(1)$ are hard to implement in retrospective without sacrificing binary compatibility to existing libraries and systems software.
3. Mapping continuation passing to a sequence of tail calls via C pointers will not work without addressing and solving the problems imposed by 1. and 2., respectively. Albeit, optimizing *indirect* tail calls turns out to be even more challenging than direct tail calls (see [3]).

Despite a possible first impression that the tail call problem might be merely a minor optimization candidate for a compiler zealot with too much time, the problem is of a very fundamental nature today, because generating code which does not comply to the C standard calling convention results in hard-to-resolve issues of binary compatibility to already existing executables as well as in portability problems. However, efforts to tackle the tail call problem, especially with respect to making GCC a better back end for declarative languages have already produced measurable improvements in that area (see § 7, [3]).

As already mentioned in § 6.1, GHC avoids the problem by letting the GCC back end emit unoptimized machine code which is then processed by a Perl script called "The Evil Mangler" [13]. This script alters the functions' *epilogues* and *prologues* such that they do not reserve stack frames for tail calls anymore. This is possible due to GHC's internal stack management

which, essentially, substitutes the platform's runtime stack in terms of function parameter passing; calls are then argument-less.

### 6.3 Garbage Collection

Rather than using `malloc` and `free` to obtain and reclaim memory, many modern programming languages, such as Java, or C#, offer the concept of garbage collection (GC) to automatically reclaim unused memory segments in the heap. In a nutshell, this is supposed to help avoiding dangerous and hard-to-find bugs like buffer overflows which are a number one source for remote exploits in today's systems software [16].

But also functional languages rely on GC to allow *lazy evaluation* [17] and higher order functions in support for notions such as continuation passing (see § 6.1, 6.2). Again, the problem with GC is that this concept is alien to C, thus mostly to GCC as well; although, this is not entirely true anymore: the GNU compiler for Java (GCJ) produces a library called `libgcj` which implements the Boehm-Weiser conservative garbage collector [18] as it is also employed in the free .NET implementation Mono [19], for instance. Boehm-Weiser realizes a basic mark-sweep algorithm to perform collections.

Closures to express the scope of functions, or objects as they are also needed in functional programming are the crux here. A closure, typically, is a function generated at runtime to capture information about the environment. Hence, it seems self-evident to put these on the architecture's stack. However, as closures usually exceed the lifespan of the items that are being referenced by it at a time (Think of tail call optimization as an example!) this often turns out impossible.

A valid alternative is often to create closures in the heap with the expense of having to garbage collect the space occupied by them. Indeed, this is what many declarative language compilers do nowadays. Java and C# on the other hand rely on garbage collection merely for a convenience reason; both do not support closures per se.

At the moment the Java front end is the only programming language implementation based on GCC that actually uses Boehm's integrated code directly, while all the functional and logic languages either maintain their own memory structures or, more recently, use alternative concepts like that of a "shadow stack" which is supposed to make collecting "garbage" from the platform's C runtime stack easier [20] and by doing all the required work entirely in the front end.

Additionally, even more subtle technical problems may arise, for instance, when pointers are hidden that really need to be visible to the collector, as it can happen when `memcpy` is used to copy these to unaligned memory locations, or by casting pointers to and from integers. While programmers would abstain from such a dangerous practice, it might turn out more difficult avoiding it totally in automatically generated code. A similar issue is related to cyclic data structures that reference to each other directly, or indirectly, like a circularly linked list: in that case the GC algorithm is unable to remove the referenced memory and the application can easily run out of heap space, despite a working GC.

The variety of different approaches to the problem of GC hints to the fact that there is, currently, not a standard solution to realize a 100% reliable automatic memory management as it necessary for a number of modern programming languages, even though the conservative garbage collector seems to be sufficient for some standard applications.

Essentially, all the open issues mentioned in this section leave language implementors two choices should they require GC: 1. they can try to cope with the restrictions the current conservative collector imposes, or 2. they can write their own memory management, in which case using the GCC back end does not provide additional convenience, unfortunately.

## 7 Conclusion

Unarguably, implementing a new programming language is but a trivial undertaking. However, this paper has shown that Open Source software and in particular the GCC suite are flexible and nowadays also mature enough to take on a whole variety of different (and sometimes tedious) tasks programmers used to tackle manually, e.g. by re-implementing well known algorithms.

Although, the presented tools can be a tremendous help when targeting a wider range of platforms, problems using this approach remain — as is sketched in § 6 — especially for declarative, or strictly object oriented programming languages.

This may be the reason why recent developments in the GCC community are largely sparked by the internals of the Java front end. The Java group not only made clear there is need for sophisticated garbage collection, but it also introduced the foundations for a new intermediate program representation that, in a sense, surpasses the expressiveness of former GCC trees.

The new representation is called GENERIC and, in the future, each front end will be required to lower any kind of program representation to the GENERIC form [21]. Subsequently, the back end will translate GENERIC into a well defined subset called GIMPLE. The "gimplifier" is necessary, because most of the lower level optimization passes are going to be defined on the *static single assignment* (SSA) form [22, 23] which can be distilled from a GIMPLE representation at ease.

SSA has been chosen, mainly because many of the newer compiler optimizations are defined over this form [24, 25, 26]. In fact, some of the new as well as old optimizations are hard to realize on trees, or RTL alone: trees tend to be rather individual for each language front end and are highly context dependent (see § 4.3), whilst RTL is often too close to being an abstract hardware platform, e.g. by associating objects to virtual stack slots rather early, although in later compilation passes these objects may have turned as redundant temporaries.

It is obvious that the SSA based rewrite of GCC is a very ambitious project that primarily affects large parts of its very complex back end. Therefore, results are not likely to ship anytime soon, at least not before GCC 3.5 is released. The current CVS version, however, works reasonably well already and effort is currently being put into porting the existing front ends over to GENERIC as is the case with Fortran 95.

However even at present, GCC is well suited as a portable back end for a variety of different programming languages. The tree representation is flexible as well as extensible (see § 4.3, 5.3) and various optimizations are performed on the existing intermediate program representations.

Unfortunately, the GCC interfaces have always changed in subtle ways and documentation on building and integrating front ends has always been sparse. By showing the most fundamental coherences between a language's grammar, the employed tools (see § 3, 4), the tree representation and the transition over to RTL (see § 5), this paper sketches all steps necessary to build a GCC based compiler, more or less, from scratch.

On the other hand, the paper has also shown where the current problem spots are, in particular, for strictly object oriented languages, or declarative ones (see § 6): although improving, the memory management of the back end and the generated code is often insufficient to deal with features like closures, garbage collection, or higher order functions.

Along with the extremely helpful GCC mailing list (archives), the existing front ends and their respective documentation, it should now be possible for anyone interested to understand what is required to integrate a new front end into the GCC suite.

Alternatively, of course, a user may chose to implement an optimizing and specialized back end himself, however, one must not forget that, despite all trouble spots, GCC contains hundreds (if not more) man-years worth of code optimizations, tweaks, and ported platforms and it seems only natural to make use of these achievements whenever possible, or even better, to help overcome problems and lead projects like the SSA rewrite to a success.

## References


[1] GNU Compiler Collection Internals. http://gcc.gnu.org/onlinedocs/gccint/.

[2] A. Aho, R. Sethi, and J. Ullman. *Compilers — Principles, Techniques, and Tools*. Addison Wesley Higher Education, 1986.

[3] A. Bauer. Compilation of Functional Programming Languages using GCC — Tail Calls. Master's thesis, Institut für Informatik, Technische Universität München, Germany, 2003.

[4] Chess/Checkers. http://www.retarget.com/.

[5] MLRISC. http://cs1.cs.nyu.edu/leunga/www/MLRISC/Doc/html/.

[6] Little C Compiler. http://www.cs.princeton.edu/software/lcc/.

[7] Zephyr/VPO. http://www.cs.virginia.edu/zephyr/.

[8] T. Conway, F. Henderson, and Z. Somogyi. Code generation for Mercury. In *Proceedings of the 1995 International Symposium on Logic Programming*, pages 242–256, Portland, Oregon, 1995.

[9] The Glasgow Haskell Compiler. http://www.haskell.org/ghc/.

[10] B. Kernighan and R. Pike. *The UNIX Programming Environment*. Prentice Hall, New Jersey, 1984.

[11] Z. Weinberg. A Maintenance Programmer's View of GCC. In *Proceedings of the 2003 GCC Developers' Summit*, pages 257–268, May 2003.

[12] The Haskell 98 Report. http://www.haskell.org/onlinereport/.

[13] The Glasgow Haskell Compiler Commentary — The Evil Mangler. http://www.cse.unsw.edu.au/~chak/haskell/ghc/comm/.

[14] W. D. Clinger. Proper tail recursion and space efficiency. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 174–185, 1998.

[15] *System V Application Binary Interface/Intel386 Architecture Processor Supplement*. The Santa Cruz Operation, Inc. (SCO), fourth edition, 1996.

[16] D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. In *Proceedings of the 2001 USENIX Security Symposium*, pages 177–190, 2001.

[17] P. Wadler. The essence of functional programming. In *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–14, Albequerque, New Mexico, 1992.

[18] H. Boehm. A garbage collector for C and C++. http://www.hpl.hp.com/personal/Hans'Boehm/gc/.

[19] The Mono Runtime. http://www.go-mono.com/runtime.html.

[20] F. Henderson. Accurate garbage collection in an uncooperative environment. In *Proceedings of the third international symposium on Memory management*, pages 150–156. ACM Press, 2002.

[21] D. Novillo. Tree SSA — A New Optimization Infrastrcutre for GCC. In *Proceedings of the 2003 GCC Developers' Summit*, pages 181–195, May 2003.

[22] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 12–27. ACM Press, 1988.

[23] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–11. ACM Press, 1988.

[24] R. Hasti and S. Horwitz. Using static single assignment form to improve flow-insensitive pointer analysis. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 97–105. ACM Press, 1998.

[25] E. Stoltz, H. Srinivasan, J. Hook, and M. Wolfe. Static single assignment form for explicitly parallel programs: Theory and practice, 1994.

[26] A. Leung and L. George. Static single assignment form for machine code. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 204–214, 1999.

# KDE Kontact: An Application Integration Framework PIM Components Get Together

David Faure
*KDE Project*
faure@kde.org

Ingo Klöcker
*KDE Project*
kloecker@kde.org

Tobias König
*KDE Project*
tokoe@kde.org

Daniel Molkentin
*KDE Project*
molkentin@kde.org

Zack Rusin
*KDE Project*
zack@kde.org

Don Sanders
*KDE Project*
sanders@kde.org

Cornelius Schumacher
*KDE Project*
schumacher@kde.org

## Abstract

Kontact is the new integrated KDE personal information management application. Well, it seems new, but in fact only the shiny surface is new. Under the hood well-known time-honored KDE applications like KMail, KOrganizer, KAddressBook, KNotes and KNode do their work. This paper describes how KDE component technologies like KParts, DCOP or XMLGUI enable embedding of applications to create something which is more than the sum of the parts.

## 1 Introduction

Kontact [1] is a new KDE [2] application for managing personal information like mail, appointments, todo lists and contacts. It is based on existing KDE applications which are embedded into a container framework by using KDE component technology and extending it where required. Figure 7 shows a screenshot of the organizer component in Kontact.

The functionality of Kontact emerges from integration of the components on the application level. It doesn't compromise the ability of the components to run as individual stand-alone applications. The user retains the choice whether to use the components as one application aggregating all the functionality or to use some or all parts as separate applications.

KDE is one of the major desktop environments for Linux and Unix systems. it is based on the Qt toolkit [29] and mostly written in C++. It consists of a framework providing the desktop infrastructure and a wide range of applications, from web browser and mail client, through music players, games and educational software to text editor and integrated development environment. Kontact is part of the kdepim module of KDE which contains tools for personal information management.

After a short discussion about components and monolithism (section 2) this paper will give an introduction into the Kontact component model (section 3) and a detailed overview of the integration technologies used in Kontact (section 4). It will discuss the importance of open standards (section 5), give an insight into Kontact as project (section 6) and will conclude with information about the availability of Kontact (section 7), final remarks (section 8) and some information about the authors of Kontact (section 9).

## 2 Components vs. Monolithism

Why bother with integrating originally disparate applications? Why not build a monolithic integrated application from ground up with all its parts fitting together from the beginning?

The technical advantages are that a loose coupling forces modularization and cleaner interface definitions. This leads to a better factored architecture which is inherently easier to maintain and leverages reuse of existing code.

On an economical side this kind of reuse saves the investments, in particular of time and effort of the open source community, in the already existing applications. All the components integrated into Kontact have a long history as independent applications and provide a stable and feature-rich base for the integrated framework.

There are also social reasons. Each of the different Kontact component applications have their own healthy developer communities. By providing a technical integration framework on the top of those applications the developers also get integrated. They still maintain their application development environment but gain a new sense of community on a higher level. The Kontact experience shows that this works surprisingly well and gives room for new synergies. The monolithic approaches we have seen in the past in this area have suffered from the not-invented-here syndrome and reinvention-of-the-wheel scenarios. There are several projects which tried to write a monolithic application comparable to Kontact and failed because of these reasons.

On a philosophical level the integration of GUI desktop applications as done by Kontact can also be seen as

interesting parallel to the classic philosophy of UNIX commands, single-purpose commands which can be assembled to complex and powerful aggregations to fulfill almost any task imaginable. By combining single-purpose applications in an integrated suite for personal information management this philosophy is raised to the application level.

# 3 The Kontact Component Model

The Kontact framework acts as a container for plugging in other applications as components. It provides the standard environment like main window, menu, tool and status bars as well as a navigation bar to control the embedded components. Applications to be embedded into this environment have to be accompanied by a Kontact plugin. The plugin acts as mediator between the framework and the application. It exposes the functionality of the application which is being integrated by implementing the Kontact Plugin API and interacts with the Kontact Framework by using the Kontact Core API. All this is done in-process. Communication between the components is done via DCOP through clearly defined standard interfaces. Figure 1 shows the component model.

The plugin API specifies functions which have to be implemented by concrete plugins for providing access to the corresponding KPart object, the summary view and options how the component appears and is handled in the framework application. This includes title labels, icons and hints about the position in the navigation bar. In addition to that it provides functions to control the component behavior, starting and selecting of a component, requesting if the component runs embedded or stand-alone and access to the interfaces of the underlying component technologies. Finally the plugin API provides functions for handling of drag and drop functionality and user interface actions.

The framework core API which gives the plugins access to the framework mainly deals with loading and selecting of plugins belonging to other components. Inter-component communication is done by using the DCOP interfaces of the individual components.

A Kontact plugin can provide an application or document main view, a summary view and some specific functionality and data like menu and toolbar actions, configuration, "about" data and more. If an application already provides a KPart as part of its interface adding a Kontact plugin in order to integrate it into Kontact is only a few lines of code.

# 4 Application Integration Technologies

The KDE framework provides a rich variety of integration technologies [5]. They have been used on a component level for applications like the KDE web browser Konqueror [14] and on a slightly higher level in the KDE office suite KOffice [15]. Kontact constitutes the integration on the highest level - the level of complete applications.

This section will present the KDE component technology KParts (section 4.1), the desktop communication protocol DCOP (section 4.2), the user interface description framework XMLGUI (section 4.3), the integrated configuration framework (section 4.4), and the KDE resources framework KResources (section 4.5).

## 4.1 KParts

KParts is the KDE component technology introduced with Konqueror and KOffice. A KPart is a dynamically loadable module which provides an embeddable document or control view including associated menu and toolbar actions. A broker returns KPart objects for certain data or service types to the requesting application. KParts are for example used for embedding an image viewer into the web browser or for embedding a spread sheet object into the word processor.

KPart instrumentation of an application is a low-effort task, because all the technologies used are usually already utilized in the existing code. There are no new programming languages, external processes or protocols involved. It basically boils down to the implementation of a specific C++ interface. Kontact components all have a KPartification history of heroic one-weekend or even one-afternoon hacks.

### 4.1.1 What's a KPart?

KParts consist of a view object embeddable in a user interface frame, associated actions like menus items and toolbar buttons and optional extension objects for handling the status bar or special functionality like the forward, backward and history functions of a browser. They are accompanied by metadata like author and copyright information and service types the KPart provides. The metadata is provided as files adhering to the Desktop Entry Standard [35]. This standard, hosted by freedesktop.org, is used in many other places in KDE and other desktop environments, e.g. GNOME [3], to store metadata.

The view object provided by KParts is a standard user interface component which is usually embedded as main component into the main window of the embedding application. It represents the view or main work area of the functionality provided by the KPart.

In addition to the view the KPart provides user interface actions which correspond to menu items or toolbar buttons. The menus and toolbars of KParts are combined with the menus and toolbars of the main window or other components. The actions are described and created based on an XML descriptions of the available actions. This mechanism is known as XMLGUI. It is de-

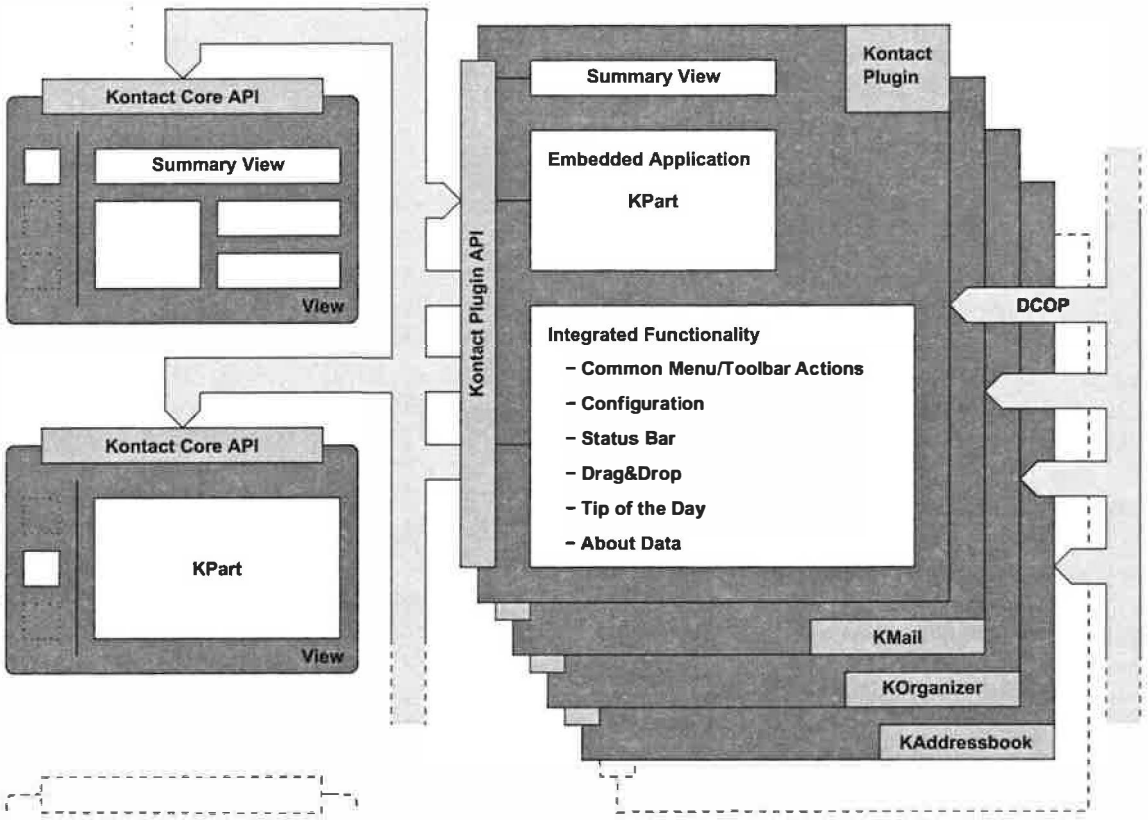



*Figure 1: Kontact Component Model*

scribed in more detail in section 4.3.

Special functionality of the embedding application for use by the KPart can be provided by extension objects which are created by the KPart and made externally visible. One example is the status bar extension which gives KParts access to the status bar of the main window of the embedding application. The KPart sends the messages to be shown in the status bar to the extension and the embedding application takes care of showing them and synchronizing the status bar information when different KParts are activated.

### 4.1.2 KParts Framework

KParts are loaded as dynamically loadable modules at run-time. A trader mechanism based on the service types provided by the KParts is used to select the KPart which fits best to the data or application to be handled and the preferences of the user. For example in Kontact this mechanism is used to find the applications which provide the Kontact integration, and in Konqueror it is used to find appropriate viewer and editor parts for handling documents of specific MIME types.

The actual KPart API basically consists of the functions to select, load and find the desired KPart instance. Specific functionality of the KPart instances is realized by subclassing. The document handling KParts used in applications like Konqueror, for example, provide an interface to load and save documents in a network-transparent way in the classes `ReadOnlyPart` and `ReadWritePart` [7].

Communication between KParts is done via the Desktop Communication Protocol (see section 4.2). This extends the API of the specific KParts by functions not only available to the embedding framework but also available to other components within or out of the process of the embedding application.

The KPart interfaces don't include remote function calls, unlike many other component technology frameworks like DCOM [8] from Microsoft, UNO [10] from OpenOffice.org or Bonobo [11] from GNOME. Historically the predecessor of the KParts framework which was based on CORBA [9] from The Open Group did so, but due to its complexity it wasn't accepted by application developers and it was eventually replaced by the current combination of KParts and DCOP which provides a simpler framework for use within a desktop environment. For a more detailed description of the KParts component architecture including a comparison with other models, especially CORBA based models see [6].

### 4.1.3 Use of KParts in Kontact

The classical KParts model originates from document-oriented views in KOffice and Konqueror. The difference in Kontact is that the KPart instances are not associated to specific documents, but represent views to more global objects, like the user's mails, calendar or contact data. Kontact doesn't subclass the KPart's API, but uses DCOP calls to access the specific functionality of the embedded applications.

Kontact loads a KPart object for each application it embeds. The KPart provides the main view which is usually used in the main window of the stand-alone version of the application. Loading is done on demand, so that only those objects consume memory and affect startup time which are actually used. Unlike in purely document-oriented user interfaces multiple KPart instances are loaded in parallel and communicate with each other, reflecting the situation of multiple applications running in parallel and working together. Concerning the user interface there still is a primary KPart instance which is active. This means that its view is shown in the main window and its part-specific actions are merged into the applications menus and toolbars.

## 4.2 Desktop Communication Protocol (DCOP)

The Desktop Communication Protocol (DCOP) is the well-known KDE inter-process communication mechanism. It is based on a central server relaying function calls between applications. DCOP makes use of the Qt object serialization implementation and uses the Inter Client Exchange (ICE) protocol [12] (part of X11R6) as transport layer.

The key feature of DCOP that makes it a viable choice for Kontact's inter-component communication is that it is also able to make in-process calls and thus minimizes the overhead for communication inside the one-process Kontact component assembly.

As this is done completely transparently to the sender and receiver of DCOP calls, it makes it possible to either run Kontact components inside of the Kontact framework or as stand-alone applications without any difference to the user other than the additional GUI integration inside the Kontact framework and the features this inherently adds, e.g the integrated configuration, the summary view or the extended drag and drop support. This gives users the choice to run the application in a way that best fits to their personal working style.

### 4.2.1 DCOP Implementation

DCOP communication between processes is mediated by a special server process. This server is started with each desktop session and handles all DCOP communication between applications belonging to this session. When an application wants to talk to another application it sends the request to the DCOP server which in turn forwards the request to its destination. Responses are sent back to the server which returns them to the application from which the request originated. The fact that communication goes through a server is completely transparent to the applications using DCOP.

When an application provides a DCOP interface for use by other processes it registers itself with the application name with the DCOP server. In addition it registers each DCOP interface it provides with a specific name with the server. DCOP provides inspection functions so that it is possible to browse DCOP interfaces in order to find out which interfaces are registered and which functions they provide.

Direct function calls are not the only way to use DCOP. it is also possible to use a signaling mechanism where an application registers for a specific signal and the server then notifies the application when the signal is emitted by the application which was requested. By using wild card matching flexible control over which signals an application listens to is possible.

Actual communication in DCOP is done using the ICE library, part of the X server, as transport layer. The communication is done by using Unix sockets. DCOP intentionally doesn't provide network transparent communication because this isn't required in normal desktop scenarios and would add an additional level of complexity as well as imposing a new class of security problems which don't need to be handled if network access is disabled.

Serialization and deserialization of data is done by using the C++ stream operators associated with all the basic data types in the Qt toolkit. They use a compact binary format. This is hidden from the DCOP user. Only if new or custom datatypes are to be sent over DCOP do new stream operators have to be implemented. This is mostly very easy because it can be based on existing operators for the data types the new type is composed of.

**DCOP interface compiler** For convenient instrumentation of an application with DCOP interfaces there is a special tool, the dcopidl compiler. It takes a file describing the interface and generates the C++ code required to implement the interface. The interface description file is basically a C++ class header with some additional keywords to identify the functions for instrumentation by DCOP. So usually it is enough to add these keywords to the already existing header declaring the functions to be available via DCOP and run the dcopidl compiler on this file. For normal compilation the special keywords are defined to be empty so that they are ignored by the compiler. Figure 2 shows an example of a header used

```
class KCalendarIface : public DCOPObject
{
    K_DCOP
  public:
    KCalendarIface()
      : DCOPObject("CalendarIface") {}

  k_dcop:
    virtual void showTodoView() = 0;
    virtual void showEventView() = 0;

    virtual void goDate( QDate date ) = 0;
    virtual void goDate( QString date ) = 0;
};
```

Figure 2: Example code providing a DCOP interface

to generate a DCOP interface.

The dcopidl compiler also provides the capability to generate stub classes which can be used to make DCOP function calls. This way the DCOP call appears as normal type-safe function call to the calling application and the DCOP communication and the fact that the call actually is a remote function call becomes completely transparent to the applications using DCOP.

The KDE build system automates running the dcopidl compiler so that DCOP interfaces can be added or used by adding only a few lines to the Makefile templates or header files.

The abstraction introduced by the dcopidl compiler could be used to replace DCOP by another transport mechanism without any changes to applications simply by modifying the dcopidl compiler to generate stubs and interface implementations for another transport mechanism. This would be one way to add support for other inter-process communication mechanisms like D-BUS [13] or platform-specific technologies.

**Application scripting** DCOP is not limited to communication between applications. It can also be used by end users to control an application remotely. There are several different methods available to do this conveniently. For example, one can use the `dcop` command line tool for sending DCOP requests, the corresponding GUI tool `kdcop`, or use the language bindings to script the application in languages like Perl or Python.

### 4.2.2 Use of DCOP in Kontact

For Kontact DCOP is the main communication mechanism between components. It is used for inter-application communication which, depending on the configuration of Kontact plugins, means interacting with applications running as external processes with their own user interface or running embedded into Kontact in-process. DCOP handles in-process calls in a special way without using the dcop server process in order to optimize the efficiency of DCOP calls.

To maximize flexibility most of the interfaces used for communication between components are not tightly coupled to the implementation of the component. There are abstract interfaces for services like "email client" or "organizer" which are implemented by specific applications like KMail or KOrganizer. In principle these services could also be implemented by other applications, so that the user could choose the specific application for being used in Kontact. This might even be used to integrate non-KDE applications by adding a small DCOP wrapper around the specific interfaces of the application to be integrated.

As components are loaded on demand and stand-alone applications might not yet have been started when another component requests communication there is a special service for starting DCOP services on demand. The component is then started according to the users configuration inside Kontact or as stand-alone process when a request to the service interface implemented by the component is initiated.

One problem that arises with components being optionally able to run stand-alone or embedded into the Kontact framework application is that the name of the application registering the DCOP interfaces is different for these two cases. Thus components inside of Kontact register twice, once with the name of the container application and a second time with the name of the stand-alone application, so that the interfaces are always available under the same name.

**Unique Applications** DCOP is also used for realizing so-called "unique applications". These are applications which can only be started once per session. This is for example used to make sure that processing of mail folders, calendar or address book data always takes place in a single process in order to avoid problems with concurrent access to the same data by different instances of an application. A unique application has a simple standard DCOP interface. When a unique application is started it first looks if a process already has registered this standard DCOP interface under the name of the application. If the process already exists it is called to handle the request to start another instance. This usually means that an existing main window is activated or that command line options are processed. If the process doesn't exist yet application startup proceeds and registers the DCOP interface, so that it is visible for subsequent starting of application instances.

The standard unique application handling isn't completely sufficient for the case of Kontact, because an application can be run in two ways, as stand-alone application where the standard unique application handling applies and as embedded component where the process providing the unique DCOP interface is different

from the stand-alone case. A DCOP watcher class handles this situation by forwarding the unique application DCOP requests to the correct destination. This makes sure that for example when calling KOrganizer from the command line and a KOrganizer component in Kontact is already running the component in Kontact is activated instead of running a second instance stand-alone.

**Hidden DCOP functions** Kontact also makes use of a special feature of the dcopidl compiler which allows developers to hide DCOP functions from the standard interface inspection provided by the dcop and kdcop tools. This makes it possible to hide parts of the DCOP interfaces, so that they can be used for inter-component communication, but are not available to tools like dcop and kdcop which rely on the interface inspection capabilities of DCOP.

## 4.3 XMLGUI

The integration of separate applications into a common main window requires merging of menus and toolbars. KDE provides an abstract way to define menu and toolbar actions using XML descriptions of where the actions are placed. These descriptions are loaded and processed at run-time. This allows to change menus and toolbars to be changed without changing code. It also provides a way to manipulate menus and toolbars programatically from outside the code implementing the actions.

This flexibility is a key requirement for embedding actions from different applications in a single framework. It allows to menus to change dynamically and toolbars to reflect the functionality provided by the currently selected component. In addition to that it allows components to inject additional actions which can optionally be available independently of the selected component and it makes it possible for the framework to remove actions which would be redundant inside the integrated application. Examples are configuration options, "new" actions and "about" dialogs, which are all provided by global actions of the container.

The menu and toolbar parts of KParts (see section 4.1) are based on XMLGUI. Figure 3 shows an example of an XML user interface action description.

One advantage of the XMLGUI mechanism is that it makes it possible to provide a general configuration mechanism for the actions to the user. There is a standard configuration dialog which enables the user to persistently change which actions are shown as toolbar buttons.

Another advantage is that a it is possible to change menus and toolbars of an application by just changing or providing more specific action descriptions. This can be used by system administrators to lock down applications and make parts of the functionality unavailable to users.

```
<!DOCTYPE kpartgui >
<kpartgui version="15" name="kontact" >
  <MenuBar>
    <Menu name="file">
      <text>&amp;File</text>
      <Action name="action_new"/>
      <Separator/>
      <Action name="file_quit"/>
    </Menu>
    <Menu name="settings">
      <text>&amp;Settings</text>
      <Merge append="save_merge"/>
      <Action name="settings_configure" />
    </Menu>
  </MenuBar>
  <ToolBar name="mainToolBar">
    <text>Main Toolbar</text>
    <Action name="action_new"/>
    <Merge/>
    <Action name="help_whats_this"/>
  </ToolBar>
</kpartgui>
```

Figure 3: Example XML description of user interface actions

This is used by the KIOSK mode described in section 4.4.3.

## 4.4 Integrated Configuration

Configuration of Kontact is done by extending the modular mechanisms used in the desktop-wide KDE control center to the application level. Configuration dialogs are composed of modules which are dynamically loaded at run-time. A generic configuration dialog provides access to all modules relevant for the applications embedded into Kontact.

The selection of the components which are present in Kontact is done by an extension of the metadata associated with all programs. The generic KDE plugin selection infrastructure makes use of this data by providing a backend for accessing activation state of plugins and a GUI for the user selecting which plugins should be active.

General access to the actual configuration options is performed through the KDE configuration backend (KConfig XT, see section 4.4.2). It provides an abstract description of configuration options which is reused in the GUI, a generated API to the configuration data which is used to access and share configuration data in a well-defined way, and other things, like the desktop lockdown features of the KIOSK mode.

### 4.4.1 Common Configuration Dialog

The configuration dialog is one example where application integration in Kontact is more than just merging views in a common main window. The configuration dialog provides access to all options of the components se-

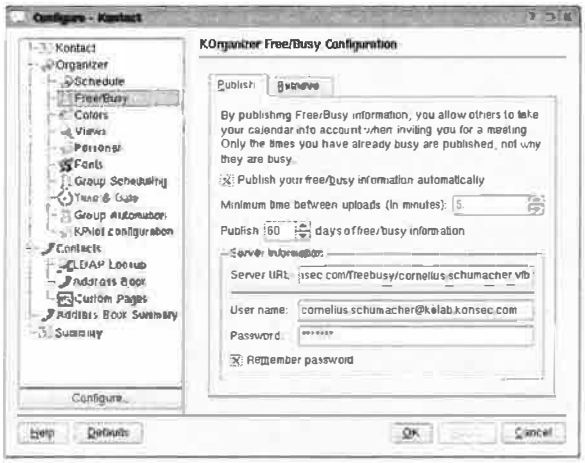



Figure 4: The Kontact configuration dialog

lected to be embedded into Kontact. It creates a common hierarchical view of all available configuration pages. See figure 4 for a screenshot of the integrated configuration dialog.

Technically the different pages of the configuration dialog are implemented as dynamically loaded modules using the same programming interfaces as the desktop-wide KDE control center. Each module has an associated file which contains meta and control information like the name of the module including translations, the name of the library to load or the type of control module, i.e. in which dialog and where in the hierarchy it is to be shown. This file uses the Desktop Entry Standard [35]. This mechanism allows the construction of the navigation part of the dialog without actually loading the modules. The fact that the modules are implemented as independent modules allows them to be loaded on demand as they are needed by the user interface, even without requiring to have the KPart of the associated application to be loaded in memory.

### 4.4.2 KConfig XT

For storage of configuration data the standard KDE configuration backend is used. It stores the data as grouped key-value pairs in simple files using an INI-style syntax. The configuration backend supports cascading configuration files, thanks to which users settings are read and merged from several files, starting from global system-wide files down to user-specific files in the users home directory. In general information from more specific files takes precedence over information from more global files, but this can be modified in the KIOSK mode.

KConfig XT (XT stands for "extended technology") is an additional level of abstraction on top of the standard configuration backend. it is based on an abstract

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE kcfg SYSTEM "http://www.kde.org/
  standards/kcfg/1.0/kcfg.dtd">
<kcfg>
  <kcfgfile name="kontactrc"/>

  <group name="View">
    <entry type="String" name="ActivePlugin">
      <default>kontact_summaryplugin</default>
    </entry>
  </group>

</kcfg>
```

Figure 5: Example XML file used by KConfig XT for describing configuration options

5 for an example file. The description contains information on the names and types of configuration entries, the text used for labeling and explaining the options, default values and a logical grouping of the entries. It also contains some control information like where the configuration is stored.

**Configuration Code Generator** To provide applications with convenient and type-safe access to the configuration data KConfig XT includes a tool for generating the C++ code, required to access the data, from the XML descriptions files: the "kconfig compiler". The generated code provides direct access to the configuration entries by creating individual functions for each entry. These can be used within the application code when access to specific entries is needed. There are various options to control the generation of the code, so that it can be adapted to the specific needs of applications. See figure 6 for an example of generated code.

The generated classes also provide a generic way to access the configuration entries. This is used for widely used actions like reading and writing of the configuration data. The generated class is also used for connecting configuration dialogs to the backend in a generic way. There is a special class `KConfigDialog` which associates the widgets of a configuration with configuration entries based on the names of the entries and the widgets. This makes it very easy to create configuration dialogs because all handling of the actual data is done automatically in the backend and the developer only has to create an XML description of the configuration options and a corresponding dialog. If this is done with GUI design tools like Qt Designer [30] creating configuration dialogs can be done without having to write more than a few lines of code.

**Tools** There are some additional tools under development which make use of the generic nature of the KConfig XT framework. One is an application-independent

figuration files via KConfig XT. This allows to manipulate options which have no representation in the GUI, to edit remote configuration files or to edit the configuration of multiple applications in one go. The other tool under development is a graphical editor for the KConfig XT XML descriptions. This makes it possible to create and edit configuration without having to know about the XML format. Together with Qt designer this makes a powerful suite of tools for easy implementation of the configuration parts of applications.

**Benefits** The advantages of the KConfig XT approach are that there is only a single location where configuration keys and default values are specified, that there is type-safe access to the data for applications, that details of reading and writing the configuration, e.g. applying default values, are automatically handled and that there is a way to access the configuration in an abstract way, which is in particular useful for generic tools not tied to a specific set of configuration data, for example a generic configuration dialog.

Kontact which, by its nature of application integration framework, combines a lot of configuration options greatly benefits from the simplifications which are made possible by the use of KConfig XT.

### 4.4.3 KIOSK Mode

To fine-tune the configurability of applications and to give administrators a tool to control how users configure their applications KDE provides the so-called KIOSK mode. This is an extension of the standard configuration backend which allows to control how configuration options can be changed. This offers all the tools required for features like desktop-lockdown.

The key concept of the KIOSK mode are immutable options. Any entry of a configuration file can be made immutable by adding a special flag to its key. This means that the configuration backend won't write any changes of this entry anymore. As configuration files are read in a cascaded way an administrator can add the flag making an entry immutable in a global configuration file which the user doesn't have permissions to write. This way the option can't be changed by the user anymore.

### 4.4.4 Configuration Wizards

With the integration of different applications in a common framework, as happens in Kontact, the need for common configuration increases. There are several configuration options which are the same for different applications, like name and email address of the user, depend on each other or can be deduced from other options, like server names, addresses of certain files or services on the same server.

One example is the configuration for access to a groupware server where data like addresses of incoming and outgoing mail server or for accessing contact or calendar data can be created from knowing which kind of server is to be used together with address and login information for a specific server.

```
namespace Kontact {

class Prefs : public KConfigSkeleton
{
  public:
    static Prefs *self();
    ~Prefs();

    /**
      Set ActivePlugin
    */
    static
    void setActivePlugin( const QString & v )
    {
      if (!self()->
              isImmutable( "ActivePlugin" ))
        self()->mActivePlugin = v;
    }

    /**
      Get ActivePlugin
    */
    static
    QString activePlugin()
    {
      return self()->mActivePlugin;
    }

    /**
      Get Item object corresponding to
      ActivePlugin()
    */
    ItemString *activePluginItem()
    {
      return mActivePluginItem;
    }
};

}
```

Figure 6: Code generated from the example XML of figure 5

Configuring this information in all the different applications is cumbersome, although the fine-grained configuration is needed for tuning the applications to special needs and to be able to handle different usage scenarios. The classical solution to this problem is to provide configuration wizards which collect the required information for configuration at a centralized location.

**Wizard Rules** In Kontact this problem is addressed by a special kind of application-spanning configuration wizard based on an extension of KConfig XT for propagation of configuration options. The information which is needed to deduce the detailed configuration for the individual applications is described by a standard KConfig XT XML file and the corresponding GUI is created

on top of this. For propagation of the information to the configuration of the individual applications a set of rules is added to the KConfig XT XML file. These rules can specify simple copying of data to other configuration files or more complex conditional propagation based on other data, e.g. based on the information if a special feature is enabled or not. it is also possible to define custom rules which are accompanied by C++ code. This gives the flexibility to also handle very complex cases.

**Wizard User Interface** The wizard dialog reads and interprets the data and rules from the description files, integrates the GUI for setting the options and applies the data put in by the user according to the defined rules to the configuration of the involved applications. This mechanism makes it very easy to set up configuration wizards as the application developer can use the generic mechanisms and only needs to write code for handling of special cases. It has the additional advantage that it creates a formal specification of how the configuration is affected by the wizard which can for example be exploited in the GUI to indicate how the different configuration options depend on each other without requiring any special handling by the application developer.

Kontact provides wizards for configuring access for groupware servers like Kolab [31] or eGroupware [32]. It would also be possible to use the wizard infrastructure for setting up user profiles or to make it possible to apply policies to the configuration. It might also be useful for storing and reapplying certain configurations of individual users, e.g. when moving between different systems.

## 4.5 KResources Framework

The way an application like Kontact accesses its data shows some common patterns for different data types. For example calendar data and address book data are accessed in a similar way. It can be stored local or remote, it can be stored in text files of different formats or in databases-like systems, there can be different sources of the same kind of data, which need different configuration, data can be read-write or read-only, etc.

To address these problems in a common and consistent way and to avoid duplication of code and effort Kontact makes use of the so-called KResources framework which provides an abstract interface for management of data resources. This framework specifies an API for data resources and the user interface to configure these resources. The resource API includes functions for saving and restoring configuration, for opening and closing of resources, for loading and saving data, for naming resources and for handling write permissions. The framework also provides a management class for handling creation, modification, configuration and persistance of KResources objects and a common configuration module which operates on the abstract API of the resources and so provides a user interface for management of all different resource objects at a central place. Usually there is only one set of resources per data type which represents for example a central calendar or address book for a user shared by different applications or Kontact components.

**Resource Families** Deriving from the abstract interface there is a set of classes defining resources for a certain type of data. They are called families in the KResources framework. Currently there are families for example for calendars and address books.

For the different families there are various implementations of concrete resources, e.g. file based calendars using the iCalendar format, address books accessed via LDAP or resources accessing calendar and address book data on a Kolab server. There are also a bit more exotic resources like one that provides entries of a Bugzilla [33] based bug-tracking system as todo list in KOrganizer.

All KResources are implemented as plugins and loaded at run-time, so that memory consumption of Kontact isn't affected by unused resources and dependencies on special external libraries are isolated in the plugins without adding to the dependencies of the Kontact framework application or other components.

**Change Notification** The KResources framework also includes a mechanism to notify different instances accessing the same set of resources about changes in the resource configuration. All resource management objects running in the same desktop session communicate with each other for this purpose, either in-process or between different processes by using DCOP. So if a new calendar file is added to the user's calendar in the central resource management configuration module it automatically appears in the calendar view of Kontact, regardless of whether the calendar runs embedded in Kontact or as stand-alone application.

# 5 Open Standards

One important aspect of interoperability between applications are open standards for file formats, network protocols and other ways of interaction or data exchange. This is important for interoperation of components inside of Kontact as well as in a wider context of interoperation with all kind of applications and servers from different provenience. The fact that the standards used are open is of particular importance for Free Software projects like Kontact to ensure that specifications and other information is available to all developers and the resulting code is free for distribution under Free Software licenses.

Kontact makes use of a wide variety of open standards. It implements many of the mail-related RFCs, including POP [18], IMAP [19] and SMTP [20]. The

address book component uses vCard [21][22] as storage and exchanges format, and provides support for LDAP [23] as an access protocol. Calendaring is based on iCalendar [24] and the associated group scheduling standards iMIP [26] and iTIP [25].

# 6 Kontact as Project

In addition to the technical aspects of Kontact it is also interesting how Kontact evolved as a project and how development works in terms of social, organizational and political aspects.

## 6.1 History

The history of Kontact is much longer than the history of the current project known under the name Kontact. It evolved along with the technologies it uses and together with the community around the components it integrates.

KMail and KOrganizer were part of KDE almost from the beginning, as separate applications. When development for KDE 2 began the kdepim module was introduced. This started as playground for a reimplementation of the KDE address book which then became the KAddressBook of today and for an experimental new mail client called "Empath". KOrganizer moved into the kdepim module shortly before KDE 2.0 which was released in October 2000.

The first implementation of a KParts-based framework that aimed at integrating various existing components of personal information management software appeared in the KDE CVS on March 22th 2000. It integrated Empath and KOrganizer but never got to a state where it really did something useful.

In April 2002 the initial version of the Kontact framework was imported under the name Kaplan into the KDE CVS. It was the result of a weekend-hack inspired by the very modular plugin structure of what now is KDevelop 3 [36], which was at that time a brand-new rewrite of KDevelop 2. It integrated KOrganizer and KAddressBook. Shortly after that KMail was also integrated, and the resulting combination was released as Kontact for the first time as a stable preview release.

At the beginning of 2003 after a meeting of many of the core kdepim developers in Osnabrueck, Germany, KMail was moved into the kdepim CVS module, now combining all major components of Kontact in one module. KDE 3.2, in the beginning of 2004, was the first KDE release which included Kontact. This was version 0.8 and already provided a full set of features, mainly due to the long history of its components.

Since then kdepim is working towards the next release which will be Kontact 1.0. One interesting aspect is that the development communities of the different applications Kontact integrates also got integrated and now form a robust and powerful team which, just as Kontact itself, is more than the sum of its parts.

## 6.2 The Inner Workings of the Kontact Team

Kontact is a classical Free Software project. It has contributors all over the world with different backgrounds and interests but with the common goal to work on Kontact and make it a good application.

Communication and coordination between developers is one crucial point of the project. Many different tools are used for this purpose, e.g. CVS, Bugzilla, IRC, mailing list, web pages, Wikis and more. The central place for communication is the kde-pim mailing list which has around 400 subscribers. More informal discussions as well as communication among developers working together for example during bug fixing sessions happen on IRC. Because contributors are distributed over many different time zones there is activity in the Kontact IRC channel almost 24 hours a day.

Personal meetings have become increasingly important and shown to be essential for efficient discussion of technical questions or common work on code as well as for fostering the relationships between the developers and increasing motivation and community feeling. Usually developers meet on trade shows or on the regular yearly KDE-wide meetings, but in the past year and a half dedicated meetings of the group of kdepim developers have also been established.

## 6.3 Commercial Improvement System

Most contributors of Kontact do their work in their free time without getting paid for it. This inevitably leads to conflicts with other engagements, be it the job, studying, or social activities. Getting paid for working on Kontact might alleviate these conflicts. In addition to that there are often people who would like to contribute to Free Software projects by spending some money. This is often difficult, because there is no infrastructure for receiving and distributing money.

For Kontact there is an attempt to provide a solution for this problem by offering a commercial improvement system [34]. This could be seen as continuation of the voting system of Bugzilla [33]. The concept is that people willing to spend some money for implementation of certain features in Kontact pledge an amount of money on the specific feature they would like to see implemented. Developers prioritize their work oriented at the accumulated amount of money per feature and work on the features with the highest amount. Once the feature is completed and the people pledging the money are satisfied with the result, they actually transfer the money to the developers having implemented the feature.

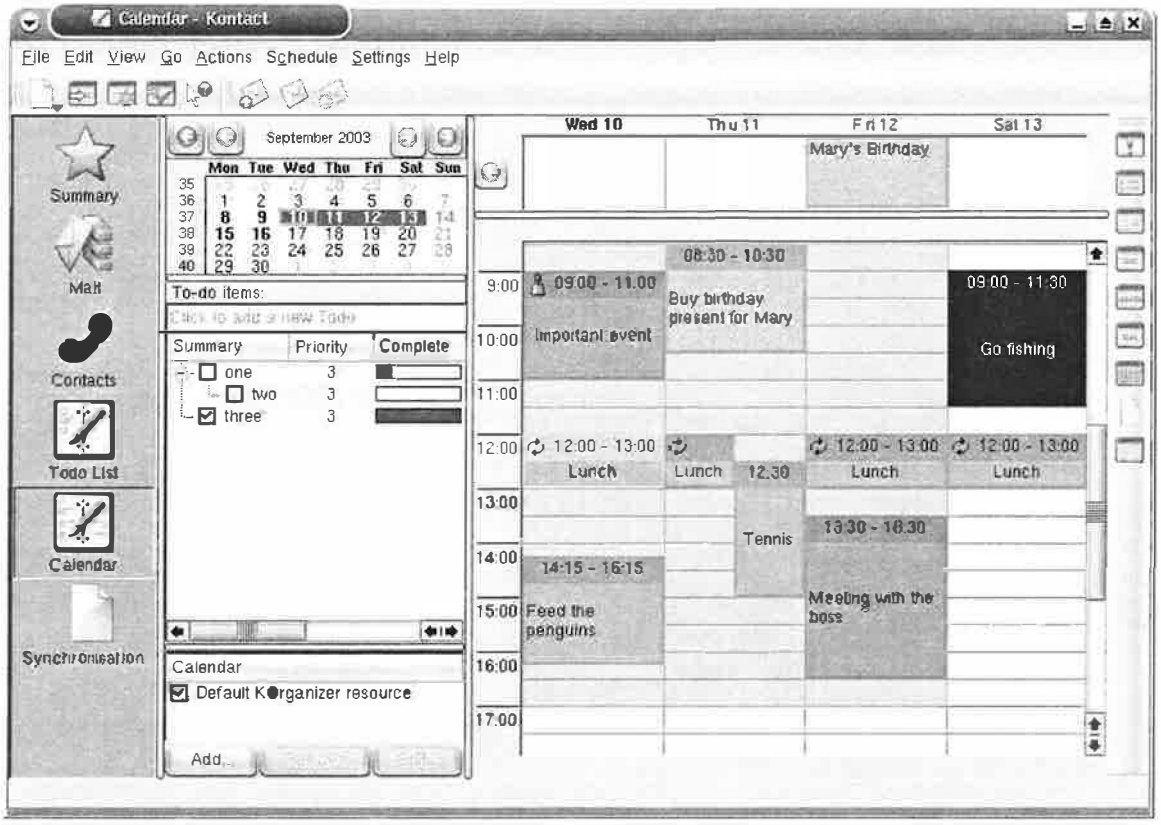



*Figure 7: Kontact screenshot showing the active calendar component*

## 7 Availability

Kontact is Free Software. Its library and interface parts are available under the GNU Library General Public License (LGPL) [28], the application itself is available under the GNU General Public License (GPL) [27].

A first stable source code release of Kontact was done in March 2003 [16]. The latest stable version was released as part of KDE 3.2 in February 2004. It can be downloaded from the KDE FTP server [17]. The next stable release (Kontact 1.0) is planned for mid of 2004. This will be the first separate release of the kdepim module independent of the other KDE modules. It will be based on the KDE 3.2 libraries.

## 8 Conclusion

Kontact introduces a new level of desktop application integration based on the technologies of the KDE framework. It provides mail, organizer, contact and other components to deliver a solution for personal information management and groupware. Its first full-featured stable release is available since the beginning of 2004.

In addition to the integration on a technical level Kontact has also integrated the development communities forming a stronger and more productive community on a higher level. This nicely demonstrates the power of Free Software development.

## 9 About The Authors

This paper was written by Cornelius Schumacher. He is a long-term kdepim contributor, one of the founders of Kontact and maintainer of several KDE applications and libraries, one of them being KOrganizer. He acts as release coordinator for the Kontact 1.0 release.

The other founding authors of Kontact are Daniel Molkentin, maintainer of the Kontact framework application, and Don Sanders, one of the founders of kdepim and co-maintainer of KMail. The original framework code was written by Matthias Hoelzer-Kluepfel.

David Faure, Tobias Koenig, maintainer of KAddressBook and Ingo Kloecker, maintainer of KMail, have done important contributions to Kontact and in particular to the Kontact application integration framework.

But the most important contributors to Kontact are the developers, translators, documentation writers and other contributors of the applications embedded into the Kontact framework. Thanks to all of them.

## References

[1] Kontact Homepage, `http://www.kontact.org`

[2] KDE Homepage, http://www.kde.org

[3] GNOME Homepage, http://www.gnome.org

[4] freedesktop.org, http://www.freedesktop.org

[5] David Sweet et. al., *KDE 2.0 Development*, SAMS (2000), http://www.andamooka.org/kde20devel.

[6] David Faure, *Coding with KParts*, http://www-106.ibm.com/developerworks/library/l-kparts/

[7] KParts API Documentation http://api.kde.org/3.2-api/kparts/html/

[8] Distributed Component Object Model (DCOM), http://www.microsoft.com/com/tech/DCOM.asp

[9] Common Object Request Broker Architecture (CORBA), http://www.omg.org/gettingstarted/corbafaq.htm

[10] Universal Network Objects (UNO), http://udk.openoffice.org/

[11] Bonobo document model, http://developer.gnome.org/arch/component/bonobo.html

[12] Inter-Client Exchange (ICE) Protocol, http://www.xfree86.org/current/ice.html

[13] D-BUS, http://freedesktop.org/Software/dbus

[14] Konqueror Homepage, http://www.konqueror.org

[15] KOffice Homepage, http://www.koffice.org

[16] Kontact 0.2.1, http://kontact.org/download

[17] KDE FTP Server, http://ftp.kde.org

[18] Post Office Protocol (POP), RFC 1939, http://www.faqs.org/rfcs/rfc1939.html

[19] Internet Message Access Protocol (IMAP), RFC 2060, http://www.faqs.org/rfcs/rfc2060.html

[20] Simple Mail Transfer Protocol (SMTP), RFC 821, http://www.faqs.org/rfcs/rfc821.html

[21] vCard 2.1 Specification, http://www.imc.org/pdi/vcard-21.txt

[22] vCard 3.0 Specification, RFC 2425, http://www.imc.org/rfc2425, RFC 2426, http://www.imc.org/rfc2426

[23] Lightweight Directory Access Protocol (LDAP), RFC 3377, http://www.faqs.org/rfcs/rfc3377.html

[24] Internet Calendaring and Scheduling Core Object Specification (iCalendar), RFC 2445, http://www.imc.org/rfc2445

[25] iCalendar Transport-Independent Interoperability Protocol (iTIP), RFC 2446, http://www.imc.org/rfc2446

[26] iCalendar Message-based Interoperability Protocol (iMIP), RFC 2447, http://www.imc.org/rfc2447

[27] GNU General Public Licence, http://www.gnu.org/copyleft/gpl.html

[28] GNU Library General Public Licence, http://www.gnu.org/copyleft/lgpl.html

[29] Qt Toolkit, http://www.trolltech.com

[30] Qt Designer, http://www.trolltech.com

[31] Kolab, http://www.kolab.org

[32] eGroupware, http://www.egroupware.org

[33] Bugzilla, http://www.bugzilla.org

[34] KDE Kontact Commercial Improvement System, http://www.kontact.org/shopping

[35] Desktop Entry Standard, http://www.freedesktop.org/Standards/desktop-entry-spec

[36] KDevelop, http://www.kdevelop.org

# mGTK: An SML binding of Gtk+

Ken Friis Larsen
ken@friislarsen.net

Henning Niss
hniss@it.edu

*Department of Innovation*
*IT University of Copenhagen*
*Denmark*

## Abstract

We describe mGTK, a Standard ML language binding for the Gtk+ toolkit. Gtk+ is a graphical toolkit for the X Window System, and provides an object-oriented C language API. Since Standard ML is a mostly-functional language without object types, constructing a binding to Gtk+ is not a trivial task. In mGTK, a single-inheritance class hierarchy is encoded using SML's type system. Most of the mGTK binding is machine generated, to best utilize the limited manpower of the project.

The goal of the mGTK project is "just" to present a type-safe interface to Gtk+ for SML programmers. This contrasts with GUI libraries for functional languages, which concentrate on producing good user interfaces: there are several SML graphical user interface libraries available for this task. With mGTK, SML applications have access to the mature, complete and familiar Gtk+ user interface.

## 1 Introduction

A good Standard ML (SML) binding to Gtk+ is an advantage for both the SML community and for the Gtk+ community. The SML community benefits in a couple of ways. First, SML programmers get access to a good general-purpose graphical user interface (GUI) library with a large range of modern widgets: the SML community is sorely missing such a library. Second, it is a step in providing full access to the whole GNOME platform. This will make it possible for SML programmers to make real applications without having to invent add-hoc solutions to many standard problems, such as database access. There are also a couple of advantages of an SML binding to the Gtk+ community. First, such a binding can help open up the small, but important, market of teaching languages. Second, as SML is a radically different language than C, an SML binding will test the "interfaceability" of Gtk+. This is an important design goal for the Gtk+ developers.

### Standard ML

SML is a functional language with imperative features which is widely used for teaching and research. It is roughly on the same level of abstraction as Python and Scheme. In contrast to Python and Scheme, which are *dynamically typed*, SML is *statically typed*, like Java and C++. This means that type errors are detected at *compile time* rather than at *run time*. Despite static typing, it is not necessary for the SML programmer to explicitly provide type annotations in the program. SML features *type inference*: the compiler reconstructs type annotations as needed.

SML is one of the few languages with a formal definition. The definition of SML [10] consists of 93 pages of mathematical notation (a "big step" structured operational semantics, plus type inference rules) and English prose. The book is not meant as tutorial for the language. Rather, it provides an implementation independent formulation of SML. This formal definition means that it is possible to write substantial applications in SML that are not dependent on a specific compiler. There are also several mature SML implementations with widely different implementation strategies, ranging from byte-code interpreters with interactive *read–eval–print–loops* to aggressive whole-program optimizing native code compilers.

### Gtk+

Gtk+, The GIMP Toolkit [14], is an LGPL-licensed [6] library for creating graphical user interfaces. It works on many UNIX-like platforms, on Windows, and on Linux framebuffer devices. Gtk+ is the graphical toolkit used in the GNOME desktop environment. Gtk+ is implemented in C, with an object-oriented hierarchy of user interface elements ("widgets"). From the beginning, the Gtk+ developers have paid attention to making it feasible and practical to develop "bindings" or "wrappers" permitting Gtk+ use by programs written in programming languages other than C. See [12] for a current list of bindings.

The Gtk+ library itself only contains widgets, but it is built on a number of useful libraries, with which it is often associated. Specific libraries used by Gtk+ are [13]:

**GLib** A general-purpose utility library, not specific to graphical user interfaces. GLib provides many use-

ful data types, macros, type conversions, string utilities, file utilities, and so forth.

**Pango** A library for internationalized text handling. It provides the rendering engine for the widgets in Gtk+ that displays text.

**ATK** The Accessibility Toolkit. It provides a set of generic interfaces allowing accessibility technologies to interact with a graphical user interface. Gtk+ widgets have built-in support for accessibility using the ATK framework.

**GDK** The abstraction layer that allows GTK+ to support multiple windowing systems.

Our work, however, is concentrated on the widget set itself.

### Overview of this paper

The rest of this paper is organized as follows. Section 2 gives a brief introduction to SML. Section 3 shows a "Hello World" example using mGTK. Section 4 explains how a single-inheritance class-hierarchy, in particular Gtk+'s, can be encoded in SML's type system, while retaining type safety. Section 5 describes the way that the mGTK library is built upon the mGTK infrastructure. Section 6 describes practical matters of mGTK, such as which SML compilers are supported. Finally, Section 7 lists related work and Section 8 gives some conclusions.

## 2 Brief Introduction to SML

This section gives a brief overview of some of the main features of SML. It is not sufficient to serve as a standalone user programming guide for the language. However, it should be sufficient to get an understanding of the examples in the rest of the paper. For more information about SML, we refer the interested reader to one of the many fine textbooks [8, 11] available.

SML is a two-level language. It consists of a *core language* for programming in the small (that is, functions, data structures and algorithms), and a *module language* for programming in the large.

Figure 1 shows a small stack library implemented in SML. This example shows most of the important features of SML. The library consists of two parts: an interface description, which is called a *signature* in SML (Figure 1(a)), and an implementation module, which is called a *structure* in SML (Figure 1(b)). Informally speaking, a signature is the "type" of a structure. It specifies the declarations of the structure that are to be externally visible.

The signature is named `STACK`. Its extent is delimited by `sig ... end`. It contains five specifications: one type specification, one exception specification, and three value specifications.

The type specification `type 'a stack` states the constraints on a module that satisfies (implements) the signature `STACK`. Such a module must declare a type named `stack`: this type is parameterized. The `'a` is a *type variable*: Type variables can be instantiated to other types. This is what is meant by a parameterized type. Thus, the type of a stack of integers is `int stack`, the type of a stack of integer stacks is `int stack stack`, and so on. Type variables are at the core of *parametric polymorphism* (similar to generics in, for example, C++ and Java; see [7] for a comparison of programming languages with support for parametric polymorphism). Note that the type specification does not say anything about how a stack must be implemented.

The exception specification states that an exception named `EmptyStack` must be declared.

The first value specification says that a constant named `empty` must be declared and that this constant must have type `'a stack`. That is, `empty` is a *polymorphic* value: it can be used in contexts where an `int stack` is needed or in contexts where a `int stack stack` is needed. The next value specification states that a function named `push` must be implemented. This function takes two arguments, an element and a stack, and returns a stack. Again, we see how type variables are used to specify that `push` must work with stacks, whereas the elements can have any type. The last value specification states that a function named `pop` must be implemented, and that `pop` takes a stack as argument and returns an element and a new stack.

Figure 1(b) shows the code of the implementation, in the form of a structure declaration. The declarations states that the structure is named `Stack`, that `Stack` satisfies the signature `STACK`, and that `Stack` does not reveal any implementation details not revealed by `STACK` (the latter connoted by `:>`). The extent of a structure is delimited by `struct ... end`.

The parameterized type `stack` is implemented by an algebraic data type described by a `datatype` declaration. This declaration says that an `'a stack` is either the constant `Empty`, or is built by applying the constructor `Stack` to an element and a stack. (Constants declared by a `datatype` declaration such as `Empty` are known as constructors).

The exception declaration `exception EmptyStack` declares an exception.

The next declaration states that `empty` is bound to `Empty`. The function `push` just applies the constructor `Stack` to its arguments. The function `pop` is more interesting. This function takes a stack as argument and then uses a `case` expression to analyze its argument. (Here we have reused the name `stack`. Types and values uses different name spaces: thus, the same name can be used for both a type and a value.) If the argument is the empty stack (the constant `Empty`) then the exception `EmptyStack` is raised (thrown). Otherwise, the ar-

```
signature STACK = sig
  type 'a stack
  exception EmptyStack
  val empty : 'a stack
  val push  : 'a * 'a stack -> 'a stack
  val pop   : 'a stack -> 'a * 'a stack
end
```

(a) interface

```
structure Stack :> STACK =
struct
  datatype 'a stack =
           Empty
         | Stack of 'a * 'a stack
  exception EmptyStack
  val empty = Empty
  fun push(elem, stack) =
      Stack(elem, stack)
  fun pop stack =
      case stack of
          Empty => raise EmptyStack
        | Stack(top, rest) =>
          (top, rest)
end
```

(b) implementation

Figure 1: Simple stack library implemented in SML.

gument has been constructed by applying `Stack` to the arguments `top` and `rest`, and then a pair consisting of `top` and `rest` is returned.

Users of this library can call functions from the structure `Stack` by using "dot-notation": for example, `Stack.pop mystack`.

This small example illustrates one of the cornerstones in functional programming: new values are constructed by analyzing, composing, and sharing old values. This is in contrast to imperative and object-oriented programming, where values are copied and modified. (A new trend in object-oriented programming is to simulate a functional style. See for example [2, Item 13 and 14].)

## 3 "Hello World" in mGTK

Figure 2 shows a deliberately simple "Hello World" example using mGTK. It illustrates (1) how to get the toolkit initialized using `GtkBasis.init` (from a module containing basic Gtk+ functionality not related to specific widgets), (2) how to construct new widgets (using module `Window` for the Window widget, and `Button` for the Button widget), and (3) how to connect signals to widgets (using module `Signal`).

Even this small example shows some of the main advantages of combining SML with Gtk+. There are no type annotations in the program source. Nonetheless, the program is statically type-checked by the compiler: type errors are found and reported at compile time rather than at runtime. In the figure we use a SML construct not explained earlier: the expression `fn _ => false` denotes an *anonymous function* that returns `false` regardless of what argument is given (you can use the *wildcard pattern* _ (underscore) to ignore an argument to a function). Anonymous functions are often handy for simple callbacks, such as this one.

The construct `let val` *x* = *exp* declares the identifier *x* to be bound to the value obtained by evaluating the expression *exp*. If the only reason for evaluating *exp* is for its side effect, one can use the wildcard pattern _ instead of *x*. Expressions evaluated only for their side effects can also be sequentialized using `;`. The value `()`, the nullary tuple of type `unit`, can be used as the return value of purely side-effecting functions. Such syntacic conveniences improve the readability and quality of code.

Finally, in SML, the double-colon `::` denotes the cons operation on lists. That is, to add an element *x* to the beginning of a list *xs* we write *x*::*xs* (in contrast to C++ where double-colon is the module operator). Built-in data types such as lists make GUI programming more convenient.

## 4 Encoding of Classes

As described in Section 1, SML is a functional language without object-oriented features, while Gtk+ is designed as an object-oriented library. This mismatch makes constructing an SML interface to Gtk+ tough . The most difficult problem is how to represent the subtype relations defined by a class hierarchy in SML's type system. In this section, we discuss how to present a type-safe SML interface to the Gtk+ class hierarchy. By *type-safe* we mean that when an SML application programmer using our library makes a type-error in calling into Gtk+ (calling a undefined method on object, for instance) the SML compiler should give a type error at compile time.

```
structure HelloWorld = struct
  fun hello _ = print "Hello World\n"

  fun main _ =
      let val _ = GtkBasis.init(CommandLine.name()::CommandLine.arguments())
          val window = Window.new ()
          val button = Button.new_with_label "Hello World"
      in  Signal.connect window (Widget.delete_event_sig (fn _ => false))
        ; Signal.connect window (Widget.destroy_sig GtkBasis.main_quit)
        ; Signal.connect button (Button.clicked_sig hello)
        ; Container.add window button
        ; Widget.show_all window
        ; GtkBasis.main()
      end
end

val _ = HelloWorld.main()
```

Figure 2: Hello World in mGTK.

Throughout, we shall think of class hierarchies mainly as definitions of subtype relationships. This "confusion" of classes and types is intentional: it is standard practice in types for object oriented programming. We are able to take advantage of two properties of Gtk+ and SML. First, Gtk+ implements only a single-inheritance class system. Second, SML's type system is expressive enough to express the subtype relations of single-inheritance class hierarchies.

We present a general method of taking a given object oriented class hierarchy and encoding it in the SML type system. The properties of the resulting encoding are: each class type has a corresponding SML type; the encoding is *complete* (all typings allowed by the class hierarchy is also allowed by the encoding); and the encoding is *sound* (all typings that is disallowed by the class hierarchy is also disallowed by the encoding). The last property is also called "type safety". In type-theoretic jargon, the trick is to use parametric polymorphism and *existential types* to encode inheritance subtyping. In particular we use *phantom types* to encode the inheritance path.

Figure 3 shows a small class hierarchy with just four classes. Label and Container are subclasses of the class Widget and Window is a subclass of Container. For the specific class hierarchy in Figure 3, the properties we want to enforce with our type encoding into SML's type system are: that we should be allowed to call the `show` method on all kinds of arguments whether they are of class type Widget, Label, Container, or Window; that we should be allowed to call the `add` method on Containers and Windows, but not Widgets and Labels; that we should only be allowed to call `set_title` on Windows; and so on.

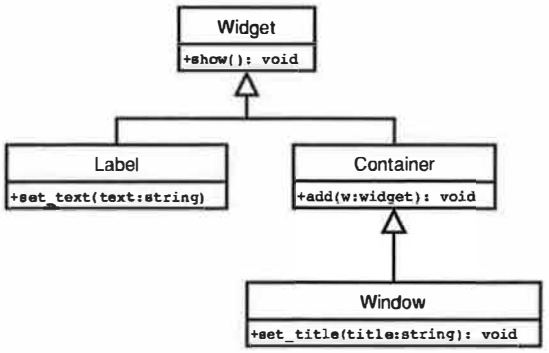



Figure 3: Small example class hierarchy

We now describe the details of the encoding of a class. Throughout this description we only present the SML specifications, that is, the parts that go into the signatures. The parts that go into the structures are not as interesting: that is simply a matter of calling into the C runtime.

**Class types:** A base class like Widget in Figure 3 is encoded as an abstract parameterized type:

```
type 'path widget
```

(We follow the convention suggested by the Standard ML Basis Library and spell type-names in lowercase, with underscores if needed.) The type variable `'path` will be used to hold the inheritance path for subclasses.

**Subtyping/Inheritance:** For a subclass like Label we need to encode two things: the existence of the class (type), and the subtype relation to the parent class. To do this, we declare two new SML types: an abstract pa-

rameterized type, and a type abbreviation for specifying the inheritance.

```
type 'path label_t
type 'path label =
        'path label_t widget
```

We call the abstract type (here `label_t`) the *witness type* because it witnesses that the class exists. Similar, the type `label` is the type abbreviation that specifies that Label inherits from Widget. In the declaration of `label` we see that the type variable `'path` in the declaration of the type `widget` has been instantiated with the type expression `'path label_t`, which contains a new type variable (also named `'path`). In the rest of the paper we shall use the convention that witness types ends with `_t`.

This is the juicy bit of the encoding, because this is really what makes it possible to encode single-inheritance class hierarchies in SML. Unfortunately, this is also the hardest part of the encoding to understand. Our experience is that you have to work a bit with some code to really comprehend the trick.

**Methods:** Because SML is not an object-oriented language we shall model methods with ordinary functions. We use the usual convention that the first argument is the object on which the method is called. (Gtk+ also uses this convention.)

We can now write the type for the method `add` in Container:

```
val add : 'path container -> 'a widget
                          -> unit
```

This specification says that `add` takes two arguments, an object of type Container and a widget, and that it returns `unit` as result. Similarly, the method `set_title` from class Window has the type:

```
val set_title : 'path window -> string
                             -> unit
```

That is, `set_title` takes two arguments, an object of type Window and a string, and it returns `unit`.

**Constructors:** We have to be a bit careful with constructors. If we return a value with a polymorphic type-variable `'path` that holds an inheritance path that has not yet been "plugged", then we could accidentally use a super-class constructor to construct values that can be instantiated to the type of a sub-class. Hence, we introduce the abstract dummy type `base` and use that to plug the type variable. Thus, the type of the constructor for Label is:

```
type base
val new : unit -> base label
```

The convention in Gtk+ is that constructors are named `new`.

**Fields:** We are not able to handle fields directly, because we keep the representation of objects completely opaque. Thus, all inspections of and changes to fields must be done through accessor methods.

We then wrap all parts of the encoding of a class into a signature/structure pair of its own. That is, for the class Window in Figure 3 the SML signature is:

```
signature Window =
sig
  type 'path window_t
  type 'path window =
      'path window_t Container.container
  val new : unit -> GtkBasis.base window
  val set_title : 'path window -> string
                               -> unit
end
```

We see that this signature relies on two structures: `Container` for the class Container, and `GtkBasis` for the dummy type `base`. In addition to the signature `Window` we also need a structure called `Window` that implements the actual calls to the relevant Gtk+ C functions.

Does this encoding really allow all the things that the Gtk+ class hierarchy allows? Yes. For example, the function `Container.add` has type:

```
val add : 'p1 container -> 'p2 widget
                        -> unit
```

From this type we can see that, if `label` is a value of type `base label` and `window` is a value of type `base window` then the application `Container.add window label` is well-typed because: (1) the type of `window` is just a abbreviation for `base window_t container`, thus, the type variable `'p1` can be instantiated to `base window_t`, and (2) the type of `label` is just an abbreviation of `base label_t widget`, thus, the type variable `'p2` can be instantiated to `base label_t`.

Consider now the function `set_title` that only works on Windows:

```
val set_title : 'p window -> string -> unit
```

If we, by mistake, attempt to use this function to set the text of the label `label` (with type `base label`) as in expression `Window.set_title label "New text"`, we get a (compile-time) type error saying (essentially) that the Label widget is not a subclass of the Window widget because the inheritance paths do not match. Here is the concrete error message given by the Moscow ML compiler:

```
- Window.set_title label "New text";
! Toplevel input:
! Window.set_title label "New text";
!                  ^^^^^
! Type clash: expression of type
!   base label_t widget
! cannot have type
!   'a window_t container_t widget
```

Hence, we have demonstrated that for these concrete examples our encoding is both sound and complete.

## 5 Process

In constructing the mGTK binding we leverage the foresightedness of the Gtk+ developers. Early on, they recognized that it would be important to have a machine-readable "specification" of the toolkit. The specification would describe the widget classes, the inheritance hierarchy, and methods and functions in the toolkit. This specification was implemented using a lisp-like custom notation in the `gtk.defs` file of the toolkit. One could argue that it is simple enough to extract the same information from the C header files. However, C headers are difficult to parse, whereas the defs format is straightforward to parse.

The bulk of the mGTK binding is constructed automatically from the `gtk.defs` file. The complete binding process is naturally divided into two phases: (1) binding design, where we apply the principles described in Section 4 to a few representative widgets to demonstrate the structure of the binding, and (2) binding construction, where the structure in (1) is applied to the entire toolkit. It is important to note here that the design phase can be carried out for a very small subset of the toolkit, after which the construction phase "mimics" that for the complete toolkit.

This phase separation makes it easier to get the design right, simply because there are fewer issues to deal with. It also makes the work involved in moving the binding to other SML compilers manageable: the compiler writers can provide the equivalent of the small subset for their compiler, and utilize that style during the construction phase. We also hope that the phase separation will help when new releases of Gtk+ are produced. Most of the work in constructing the binding for the new release is over when the design of the small subset has been completed.

Let us return to our running example, and look at some example specifications of widgets, functions/methods, and signals. Figure 4 shows three entries in the `gtk.defs` file. The first entry shows a widget specification indicated by `define-object`. From the entry we see that the `GtkContainer` widget (the name appearing right after `define-object` is a shorthand) inherits from `GtkWidget`, and it belongs in the `Gtk` module. We also see the type assigned to instances of this widget in the *Gtk+ type system* (which is completely unrelated to the SML encoding given above).

```
(define-object Container
  (in-module "Gtk")
  (parent "GtkWidget")
  (c-name "GtkContainer")
  (gtype-id "GTK_TYPE_CONTAINER")
)

(define-method gtk_container_add
  (of-object "GtkContainer")
  (c-name "gtk_container_add")
  (return-type "none")
  (parameters
    '("GtkWidget*" "widget")
  )
)

(define-signal delete-event
  (of-object "GtkWidget")
  (return-type "gboolean")
  (when "last")
  (parameters
    '("GdkEventAny*" "p0")
  )
)
```

Figure 4: `gtk.defs` excerpt.

The next entry shows a method specification for the method `add`. This method takes a `GtkWidget*` (in the C implementation) argument, and returns nothing. Since it is a method, there is an implicit "self" argument of type `GtkContainer*`.

The final entry shows a "signal handler" or ("callback") specification. In this case, we specify the prototype for handlers of delete events on widgets. The signal handler for `delete-event` for widget `GtkWidgets` accepts a parameter of type `GdkEventAny*`, and returns a value of type `gboolean`.

## 6 The mGTK Binding

The mGTK binding is available at SourceForge `http://mgtk.sf.net/` and is released under the GNU Lesser General Public License (LGPL) [6].

A fundamental difference in producing SML bindings of Gtk+, compared to bindings for other languages, is the existence of a variety of compilers (Section 1). This sets this work apart from bindings to languages such as Python, where there is only one target compiler and runtime system.

The encoding of the Gtk+ class hierarchy in the SML type system in Section 4 is *the* core aspect of the binding. As the encoding stays within the language as defined in the Definition [10], this aspect of the binding remains the same for all SML compilers conforming to the Definition. In other words, the interface exposed to the application programmer is the same across all compilers. One finds SML and Gtk+ implementations on a large variety of platforms. Thus, the GUI porting work in moving application programs from one of these platforms to another is largely eliminated.

The mGTK binding already targets two of the main SML systems, Moscow ML [17] and MLton [16]. The authors are currently looking into constructing bindings for other SML compilers (in particular, the *ML Kit with Regions* [15] and *SML.NET* [18] with Gtk#). As mentioned earlier (Section 5), the issues here mainly involve interfacing to C.

The potential for partial compiler independence sets the present binding apart from other Gtk+ bindings for SML; notably, the `SML-Gtk` binding for the SML of New Jersey compiler [9]. The `SML-Gtk` binding is also based on phantom types. Our binding predates the `SML-Gtk` binding by approximately two years—the `SML-Gtk` User's Manual refers to the mGTK binding. To date no serious attempts has been made to merge these two projects. The reason for this is that, even though the projects seems similar, we have followed rather different strategies for constructing our respective bindings. `SML-Gtk` is partly generated by the `ml-nlffi` foreign function interface, for instance, and does not attempt to automate memory management.

## 7 Related Work

The list of language bindings for Gtk+ shows a plethora of different languages from which Gtk+ is accessible. In this section we briefly discuss the bindings most related to mGTK.

When considering ML-like languages, there are two major alternatives to the mGTK binding. The `SML-Gtk` binding was discussed earlier. The `lablgtk` binding is a Gtk+ binding for O'Caml. O'Caml is a ML dialect different from SML: among other things, it has object-oriented features. This binding, therefore, can directly utilize the Gtk+ object hierarchy.

Gtk+ has also been bound to other functional languages. For example, `gtk+hs` is a Haskell binding, and `erlgtk` is an Erlang binding. Bindings also exist for other graphical toolkits. For example, `sml_tk` is an SML binding of Tk.

The use of phantom types to express invariants about programs is not new. However, the encoding of a single-inheritance hierarchy as above is original with us. Independent work has established similar results [5]. On the construction side of things, other bindings are also machine generated. For this, some of the bindings use the *Simplified Wrapper and Interface Generator* (SWIG) [1], while others extract appropriate information directly from the C headers files of Gtk+.

From the outset, the necessity of access to libraries has been realized in the functional programming community. Work in this area for SML includes SML/NJ's foreign function interface [3]; for Haskell it includes [4].

## 8 Conclusions and Future Work

It is our intention to continue this work by utilizing appropriate programming language technology to gradually bind more and more of the GNOME development platform for SML. As was the case above, this entails designing appropriate representations of the platform in the SML world (in particular, preserving the type-safety property mentioned above). It also includes the more practical work of extending the code generator to handle such newly introduced representations.

The long term goal for mGTK is to target most of the GNOME platform. The advantages of bringing GNOME to the SML community in the form of such bindings are twofold. Firstly, it would allow SML programmers access to the vast collection of useful application-level support in GNOME. Secondly, it would allow SML programmers to take part in the development of GNOME components, by allowing them to write such components in SML. The key technical aspect to be solved here is to support type-safe inheritance on the SML side of things. Of course, one will also have to explore exactly how to tie the various languages together. The GNOME community already has experience in this area.

In this paper we have demonstrated that it is theoretically and practically possible to make a type-safe interface from SML to Gtk+. This is interesting for several reasons. First, mGTK was one of the first graphical toolkits available to the SML community. Second, the fact that it is possible to make an SML binding to Gtk+ attests to the claimed "interfaceability" of Gtk+, because SML is so radically different from C in abstraction level and paradigm. Third, by auto-generating the binding, we get a binding of the complete Gtk+ toolkit. Finally, we believe that the particular way we construct the binding can be extended to bind the entire GNOME development platform, using mainly machine generated stub code.

## 9 Acknowledgments

We are grateful to the lead developer of Moscow ML, Peter Sestoft, who has answered a multitude of questions about interfacing Moscow ML with C libraries. Without Peter's initial technical support, mGTK would never have seen the light of day. Likewise, the MLton develop-

ers, in particular Stephen Weeks, has been supportive in answering questions about interfacing MLton to C. They have even made changes to the MLton compiler that were needed for mGTK. Also, we would like to thank the IT University of Copenhagen, were we have been employed while doing much of the work presented in the article. Finally, we must express our deepest gratitude to our FREENIX shepherd on this article, Bart Massey. Without his patience, encouragements, and many suggestions for improvements, this article would have been much less readable.

## References

[1] David M. Beazley. SWIG: An easy to use tool for integrating scripting languages with C and C++. In *Proceedings of 4th Annual USENIX Tcl/Tk Workshop*, pages 129–139. USENIX Association, 1996.

[2] Joshua Bloch. *Effective Java.* The Java Series. Addison-Wesley, 2001.

[3] Matthias Blume. No-longer-foreign: Teaching an ML compiler to speak C "natively". *Elec. Notes in Theo. Comp. Sci.*, 59(1), 2001.

[4] Sigbjorn Finne, Daan Leijen, Erik Meijer, and Simon L. Peyton Jones. Calling hell from heaven and heaven from hell. In *Int. Conf. on Func. Prog. (ICFP'99)*, pages 114–125, 1999.

[5] Matthew Fluet and Riccardo Pucella. Phantom types and subtyping. In *IFIP Theo. Comp. Sci. (TCS'02)*, pages 448–460, 2002.

[6] Free Software Foundation. GNU Lesser General Public License (LGPL), 1999. URL `http://www.gnu.org/licenses/lgpl.html`.

[7] Ronald Garcia, Jaakko Jarvi, Andrew Lumsdaine, Jeremy Siek, and Jeremiah Willcock. A comparative study of language support for generic programming. In *Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications (OOPSLA 2003)*, pages 115–134. ACM Press, 2003.

[8] Michael R. Hansen and Hans Rischel. *Introduction to Programming using SML.* Addison-Wesley, 1999.

[9] Allen Leung. `SML-Gtk`: Gtk+ bindings for Standard ML of New Jersey, 2003. URL `http://www.cs.nyu.edu/phd_students/leunga/sml-gtk/sml-gtk.html`.

[10] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised).* The MIT Press, 1997.

[11] Larry Paulson. *ML for the Working Programmer (2nd edition).* Cambridge University Press, 1996.

[12] The Gtk+ language bindings webpage, 2004. URL `http://www.gtk.org/bindings.html`.

[13] The Gtk+ reference manual, 2004. URL `http://developer.gnome.org/doc/API/2.0/gtk/index.html`.

[14] The Gtk+ webpage, 2004. URL `http://www.gtk.org/`.

[15] The MLKit web page, 2003. URL `http://www.it-c.dk/research/mlkit/`.

[16] The MLton web page, 2003. URL `http://www.mlton.org/`.

[17] The Moscow ML web page, 2003. URL `http://www.dina.kvl.dk/~sestoft/mosml.html`.

[18] The SML.NET web page, 2003. URL `http://www.cl.cam.ac.uk/Research/TSG/SMLNET/`.

# Xen and the Art of Repeated Research

Bryan Clark, Todd Deshane, Eli Dow, Stephen Evanchik, Matthew Finlayson, Jason Herne, Jeanna Neefe Matthews
*Clarkson University*
*{clarkbw, deshantm, dowem, evanchsa, finlayms, hernejj, jnm}@clarkson.edu*

## Abstract

Xen is an x86 virtual machine monitor produced by the University of Cambridge Computer Laboratory and released under the GNU General Public License. Performance results comparing XenoLinux (Linux running in a Xen virtual machine) to native Linux as well as to other virtualization tools such as User Mode Linux (UML) were recently published in the paper "Xen and the Art of Virtualization" at the Symposium on Operating Systems Principles (October 2003). In this study, we repeat this performance analysis of Xen. We also extend the analysis in several ways, including comparing XenoLinux on x86 to an IBM zServer. We use this study as an example of repeated research. We argue that this model of research, which is enabled by open source software, is an important step in transferring the results of computer science research into production environments.

## 1. Introduction

Repeated research is a well-respected model of investigation in many sciences. Independent tests of published research are valued because they document the general applicability of results. In addition, repeated research often sheds new light on aspects of a work not fully explored in the original publication.

In computer science, however, it is most common for researchers to report results from testing the software that they themselves have implemented. There are many reasons for this, including the wide variety of hardware and software platforms and the difficulty transferring fragile research software to a new environment. However, without independent trials, it is difficult to establish reported experience as repeatable fact.

Computer systems researchers often note with dismay the number of great ideas that are not incorporated into production computer systems. We argue that encouraging repeated research is an important step towards this transfer of technology. Researchers who release their code to the open source community make a valuable step towards encouraging repeated research in computer science.

In this paper, we present results that repeat and extend experiments described in the paper "Xen and Art of Virtualization" by Barham et al. from SOSP-03. [Xen03]. Xen is an x86 virtual machine monitor produced by the University of Cambridge Computer Laboratory in conjunction with Microsoft Research and Intel Research. Xen has been released under the GNU General Public License at xen.sourceforge.net.

In [Xen03], Barham et al. explore the performance of XenoLinux – Linux running in Xen. They compare performance to native Linux as well as to other virtualization tools such as User Mode Linux (UML) and VMWare Workstation. They also examine how the performance of Xen scales as additional guest operating systems are created.

In this paper, we first report the results of repeating measurements of native Linux, Xenolinux and User Mode Linux on hardware almost identical to that used in the Xen paper. Second, we present results comparing Xen to native Linux on a less powerful PC. Third, we evaluate Xen as a platform for virtual web hosting. Fourth, we compare the performance of benchmarks running in XenoLinux to the same benchmarks running in Linux on an IBM zServer that we won as a prize in the 2001 IBM Linux Scholar Challenge competition. Finally, we discuss our general experiences with repeated research.

We structure the rest of our paper around a set of questions and their answers.

- Can we reproduce the results from the SOSP-03 Xen paper?
- Could we realistically use Xen for virtual web hosting?
- Do you need a $2500 Dell Xeon Server to run Xen effectively, or will a 3 year old x86 do the job?

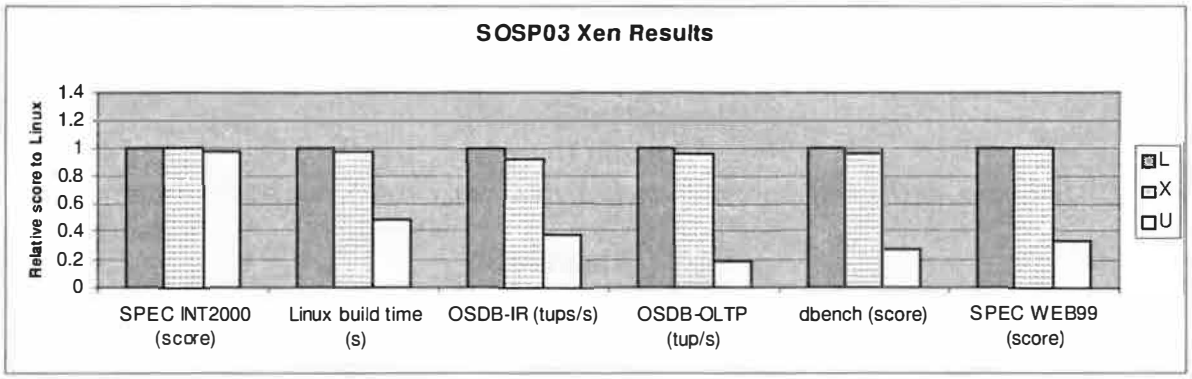
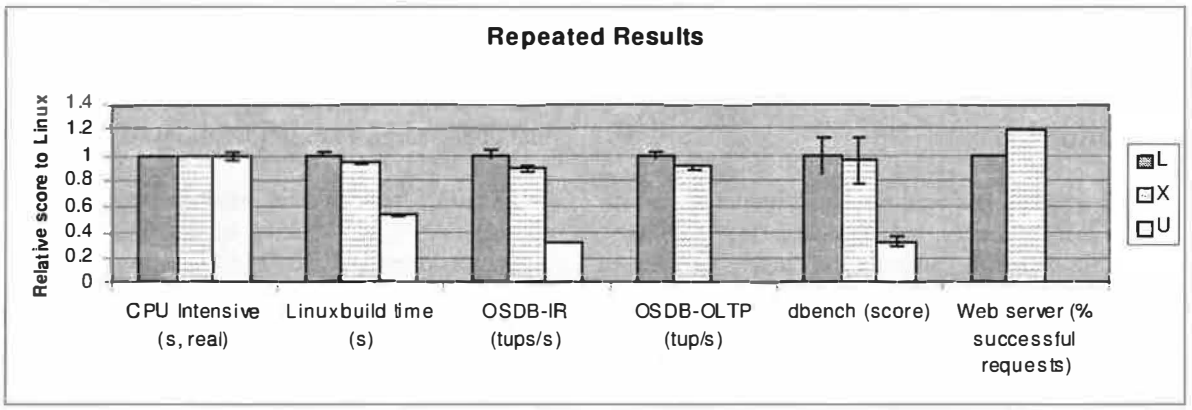



**Figure 1a Relative Performance of Native Linux (L), XenoLinux (X) and User-Mode Linux (U).** This data is from Figure 3 of [Xen03].




- How does a virtual machine monitor on commodity PCs compare to a virtual machine monitor on a mainframe specifically designed to support virtualization?
- What have we learned about repeated research?

## 2. Repeat the SOSP-03 Performance Analysis of Xen

Our first task was to convince ourselves that we could successfully reproduce the results presented in [Xen03]. The paper itself contained clear details on their test machine – a Dell 2650 dual processor 2.4GHz Xeon server with 2 GB RAM, a Broadcom Tigon 3 Gigabit Ethernet NIC and a single Hitachi DK32EJ 146 GB 10K RPM SCSI disk.

We had little trouble acquiring a matching system. We ordered a machine matching their specifications from Dell for approximately $2000. If we had been repeating older research, reproducing an acceptable hardware platform might have been a significant challenge.

The only significant difference in our system was the SCSI controller. Their controller had been a 160 MB/sec DELL PERC RAID 3Di and ours was a 320 MB/sec Adaptec 29320 aic79xx. Thus our first hurdle was the need to port the driver for our SCSI controller to Xen. The Xen team was extremely helpful in this process and in the end we contributed this driver (and several others) back into the Xen source base.

Our second hurdle was assembling and running all of the benchmarks used in the Xen paper including OSDB, dbench, lmbench, ttcp, SPEC INT CPU 2000 and SPECweb99. (The Xen team was quite helpful in providing details on the parameters they used for each test and even providing some of their testing scripts.) We generalized and extended their scripts into a test suite that would help save others this step in the future.

In our test suite, we replaced SPEC INT and SPECweb as the details of the test are not made public and they are only available for a fee from SPEC [SPECWEB] [SPECINT]. (Open benchmarks are nearly as important

as open source!) Instead of CPU-intensive SPECINT 2000, we chose FourInARow, an integer intensive program from freebench.org [FourInARow]. We wrote our own replacement for the web server benchmark, SPECweb99, using Apache JMeter. We discuss our web benchmark in more detail in Section 3.

Our final hurdle was that our initial measurements showed much lower performance for native Linux than [Xen03]. In comparing the details of our configuration with the Xen team, we discovered that performance is much higher with SMP support disabled.

With those hurdles behind us, we successfully reproduced measurements from [Xen03] comparing the performance of XenoLinux and UML to native Linux. In Figure 1, we show the results from Figure 3 of [Xen03] and our results. In Figure 1a, we show data from Figure 3 of [Xen03] (minus the data on VMWare workstation). In Figure 1b, we show our results from performing similar experiments. The native Linux results are with SMP disabled in all tests.

We add error bars to illustrate standard deviation where we ran at least 5 tests of each benchmark. OSDB on UML gave errors in the majority of runs. We received only one score for OSDB-IR and no scores for OSBD-OLTP from all our tests. We are missing some measurements for UML. We investigated further, but were unable to determine a cause.

Reporting standard deviation adds important information about the reliability of a reported score. The standard deviation of most benchmarks is less than 1%. Dbench has a standard deviation of 14% and 18% for native Linux and XenoLinux respectively.

In our tests, the relative performance of XenoLinux and UML compared to native Linux is nearly identical to the performance reported in [Xen03] as shown in Figures 1a and 1b. Our CPU-intensive and web server benchmarks are not directly comparable to SPEC INT and SPECweb99, but accomplish a similar purpose and demonstrate similar relative performance.

In Table 1, we extract only the Xen bars from Figure 1 for the benchmarks that are directly comparable: Linux build time, dbench, OSDB-IR and OSDB-OLTP. Our tests show Xen to be slightly slower relative to native Linux on three of the four repeated tests. In each case the difference is less than 5%, but it is also outside the standard deviation that we measured. Because the difference is so small in this case, we don't see a problem with the results in [Xen03]. However, it is a good illustration of the value of repeated results in validating published numbers.

Our web server benchmark shows Xen to be better than native Linux with SMP disabled. However, if we compare to Linux with SMP enabled, Xen and native Linux are nearly matched as shown in [Xen03] Figure 2. This is one frustration we had with the results in [Xen03]: some results are reported with SMP enabled and some with SMP disabled. The authors gave native Linux as much advantage as possible relative to Xen each time. This is certainly honorable, but it made repeating the results more difficult.

Finally, we are ready to answer our first question: Can we reproduce the results from the SOSP-03 Xen paper? We have mentioned a few caveats, but overall the answer is yes. We can reproduce the comparison of XenoLinux and native Linux to within a few percent on nearly identical hardware.

| Performance Relative to Native Linux | Xen Repeated (std deviation) | Xen SOSP-03 |
|---|---|---|
| Linux Build | 0.943 (0.003) | 0.970 |
| OSDB-IR | 0.892 (0.024) | 0.919 |
| OSDP-OLTP | 0.905 (0.020) | 0.953 |
| dbench | 0.962 (0.182) | 0.957 |

**Table 1 Comparing the Relative Performance of XenoLinux to native Linux in our repeated experiments to the results in the [Xen03]. Numbers represent the percentage of the original Linux performance retained in XenoLinux. Numbers in parentheses are the standard deviation.**

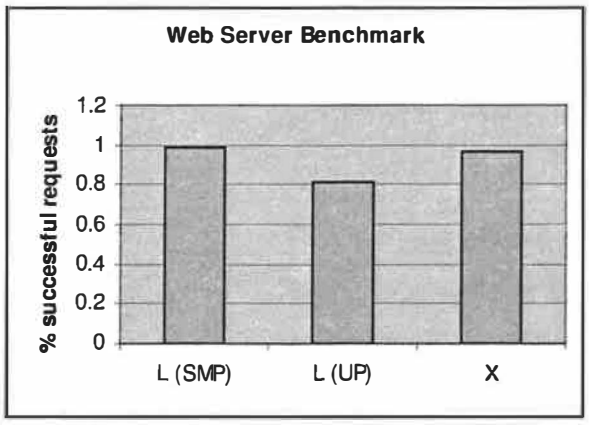



**Figure 2 Comparing Xen to native Linux with SMP enabled to native Linux with SMP disabled for our web server benchmark.**

## 3. Xen and Virtual Web Hosting

One of the stated goals of Xen is to enable applications such as server consolidation. In comparing Xen to Denali, [Xen03] page 2 states "Denali is designed to support thousands of virtual machines running network services which are small-scale and unpopular. In contrast, Xen is intended to scale to approximately 100 virtual machines running industry standard applications and services."

We set out to evaluate the suitability of Xen for virtual web hosting. Specifically, we wanted to determine how many usable guests could be supported for the purpose of hosting a web server.

[Xen03] includes a figure showing the performance of 128 guests each running CPU Intensive SPEC INT2000. We hoped to begin by showing the performance of 128 guests each running a web server benchmark. However, when we went to configure our Dell Xeon server for this test, we ran into certain resource limitations. First, as they state in the paper, the hypervisor does not support paging among guests to enforce resource isolation. Therefore each guest must have a dedicated region of memory. For the 128 guest SPEC INT test, they used 15 MB for each guest reserving 80 MB for the hypervisor and domain0 [Pratt03]. This is not sufficient for an industry standard web server. Second, they used raw disk partitions for each of the 128 guests. The Linux kernel supports only 15 total partitions per SCSI disk. Getting around this limit requires patching the kernel (as the Xen team did) or using a virtualized disk subsystem. We tried the virtualized disk subsystem in the Xen 1.0 source tree without success. We plan to evaluate the 1.1 source tree.

If we were to increase the memory allocated per guest from 15 MB to a more typical memory size of 128 MB, we could accommodate only 15 guests plus domain0. To support 100 guests at 128 MB per guest would require over 12 GB of memory. At 64 MB per guest, 100 guests would require over 6 GB of memory. In our Xeon server, the most memory we can support is 4 GB.

We also wished to re-evaluate the performance of multiple guests running concurrent web servers under load. Figure 4 of [Xen03] compares 1-16 concurrent web servers running as separate processes on native Linux to the same number running in their own Xen guest. The results indicate little degradation even at 16 concurrent servers.

Instead of using SPECweb99 to measure web server performance as in [Xen03], we wrote a replacement for it using Apache JMeter. JMeter is a flexible framework for testing functionality and performance of Web applications under load. More information including our JMeter test plans and documentation is available at http://www.clarkson.edu/class/cs644/xen.

Table 2 shows the type and distribution of requests sent to the Web servers under test in SPECweb99 [SPECWEB]. They base this distribution on an analysis of typical web server logs. We instrumented JMeter to follow the same distribution of requests and placed the proper static and dynamic content on each server.

SPECweb99 reports the number of simultaneous connections that meet a minimum set of error rate and bandwidth requirements [SPECWEB]. If a connection does not conform to the requirements, it does not contribute at all to the score. For our tests, we sent requests from JMeter engines on four remote clients. We do not factor out requests from "non-conforming" clients, nor do we limit the request rate from these machines. The tests complete after a specified number of requests have been issued. This number scales directly with the number of servers under test. We measured how many of the requests were returned correctly within 300 ms. We chose this value as a reasonable packet response time over a fast private LAN.

Due to the difference in reporting, we cannot compare SPECweb99 results directly to the results from our web server tests. Figure 3 reports our results for 1 to 16 concurrent servers. We report results for native Linux both with SMP enabled and disabled. For Xen, we allocated 98 MB for each guest in addition to domain0.

Our results show that native Linux with SMP enabled retains high performance even with 16 concurrent web server processes under high load significantly higher than SPECweb99. XenoLinux drops off steadily as more guests are added. Linux with SMP disabled is shown for completeness.

Thus, we are ready to answer our second question: Could we realistically use Xen for virtual web hosting? We have found Xen to be quite stable and could easily imagine using it for 16 moderately loaded servers. However, we would not expect to be able to support 100 guests running industry standard applications.

| | | | |
|---|---|---|---|
| 70% Static Content | | | |
| | 35% | Less than 1KB class | 9 files evenly distributed in the range |
| | 50% | 1-to-10-KB class | 9 files evenly distributed in the range |
| | 14% | 10-to-100-KB class | 9 files evenly distributed in the range |
| | 1% | 100KB-to-1MB class | 9 files evenly distributed in the range |
| 30% Dynamic Content | | | |
| | 16% | POSTs to simulate user registration forms | Generic user registration form written. |
| | 41.5% | GETs to simulate ad banner rotation | Number of banner files in SPECweb99 unknown; We used 3 files 468x60 pixels. |
| | 42% | GETs with cookies | Set a single cookie < 1K. |
| | 0.5% | CGI GETs for CGI web pages | Simple HTML page < 1K generated. |

**Table 2 Type and Distribution of Web Requests**

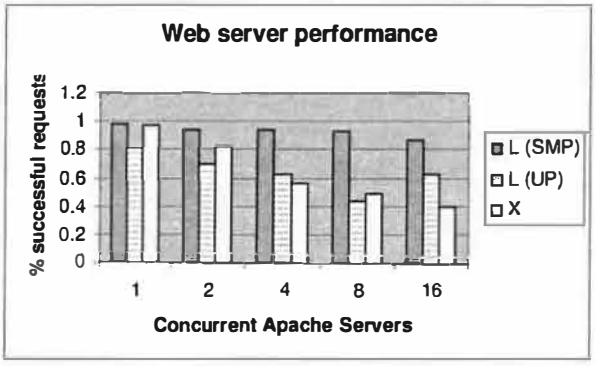



**Figure 3 Web server performance for native Linux with SMP enabled, native Linux with SMP enabled and XenoLinux.**

## 4. Comparing XenoLinux to Native Linux on Older PC Hardware

After reading [Xen03], we wondered how Xen would perform on an older PC rather than a new Xeon Server. So in addition to running on a 2.4 GHz dual processor server, we ran our tests on a P3 1 GHz processor with 512 MB of PC133 memory with 10/100 3COM (3c905C-TX/TX-M Ethernet card) and a 40 GB Western Digital WDC WD400BB-75AUA1 hard drive.

In Figure 4a, we first show the performance of Xen and native Linux on this older PC platform relative to native Linux on the Xeon server. Clearly raw performance is less on the older PC. In Figure 4b, we show the relative performance of Xen to native Linux on the older platform to the relative performance of Xen to native Linux on the faster platform. On average, Xen is only 3.5% slower relative to native Linux on the older PC.

Although the relative overhead is nearly the same on both systems, one disadvantage of the older PC is that we will be able to create fewer guests. For example, while we are able to create 16 guests with 128 MB of memory each on the Xeon server, we can create only 3 such guests plus domain0 on the older PC.

Thus, we are ready to answer our third question: Do you need $2500 Dell Xeon Server to run Xen effectively or will by 3 year old x86 do the job? No, an older PC can be used to efficiently use Xen, but only with a small number of guests.

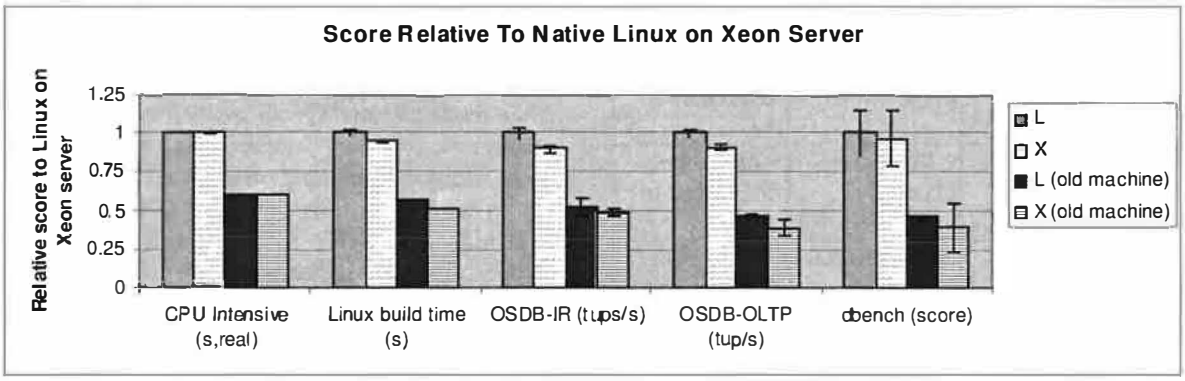



**Figure 4a Relative Performance of native Linux and Xen on new Xeon server.**

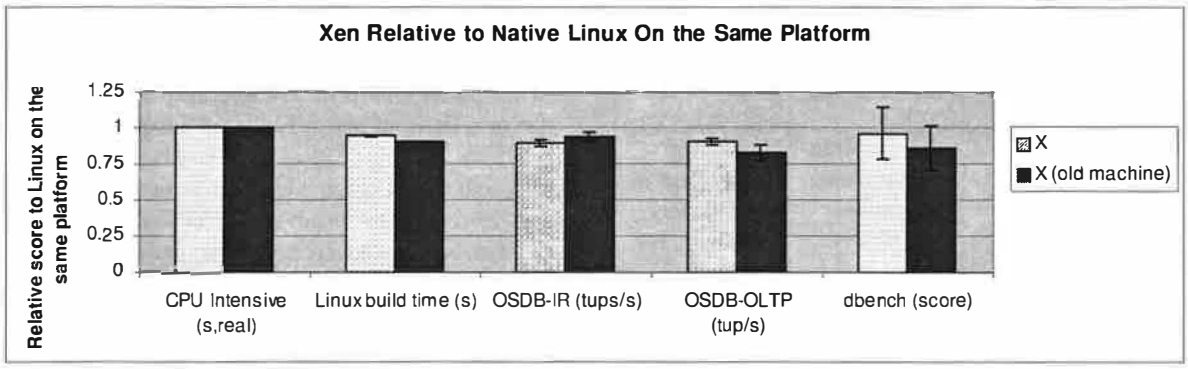



**Figure 4b Relative Performance of Xen to native Linux on the same platform.**

## 5. Xen on x86 vs IBM zServer

Virtualization for the x86 might be relatively new [Denali02, VMWare], but it has been around for over 30 years on IBM mainframes [VM370]. After reading [Xen03], it is natural to question how multiple Linux guests with Xen on x86 compare to multiple Linux guests on an IBM mainframe designed specifically to support virtualization. This is especially relevant given the following posting from Keir Fraser of the Xen team to the Linux Kernel Mailing List: "In fact, one of our main aims is to provide zseries-style virtualization on x86 hardware!" [LKML03]

In 2001, some of the authors won the top prize in the IBM Linux Challenge competition, a zServer. Specifically, we have an IBM eServer z800 model 2066-0LF with 1 processor and 8 GB of memory. It is connected to an ESS800 Enterprise Storage System via Ficon with 2 channel paths from 1 Ficon card. This machine was valued at over $200,000.

Our zServer is an entry-level model. The single CPU executes a dummy instruction every other cycle; a software upgrade is required to remove this feature. It could be configured to have up to 4 CPUs and up to 32 GB of memory. In addition, we could get up to 8 times the I/O bandwidth with additional FICON controllers

In Figure 5, we compare performance on the zServer to native Linux and Xen on both the new Xeon server and the old PC. On the zServer, we ran Linux guests with the 2.4.21 kernel just as in our x86 native Linux and Xen tests. For the zServer, it is specifically 2.4.21-1.1931.2.399.ent #1 SMP. We found that Xen on the Xeon server significantly outperforms the zServer on these benchmarks.

At first, we were surprised by these results. However, results presented by IBM in "Linux on zSeries Performance Update" by Thoss show comparable performance for a modern z900 with 1 CPU [ZPERF]. In Figure 6, we present a graph similar to one in [ZPERF, p14] showing the performance of dbench on our zServer, our Xeon server and our older x86. As in [ZPERF], throughput for a one CPU zServer does not rise above 150 MB/sec. However, we show a more significant degradation for more than 15 concurrent clients.

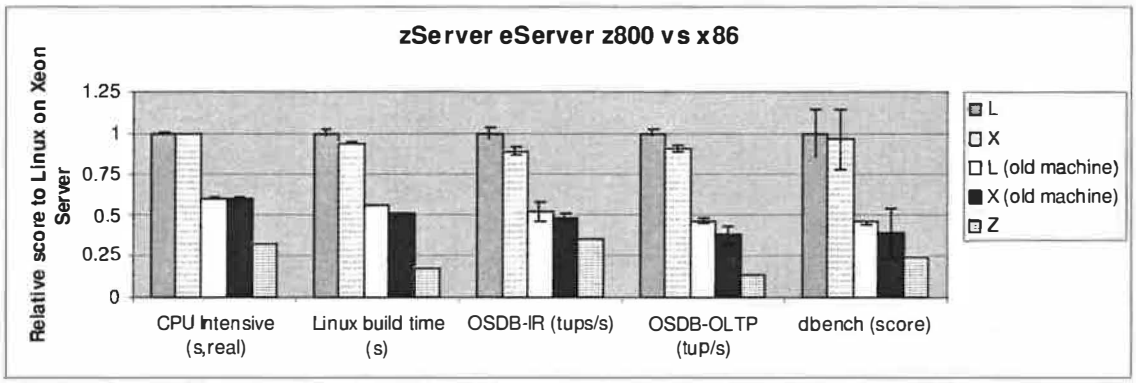



**Figure 5 Performance on the zServer shown relative to native Linux on the Xeon server; Xen on the Xeon server as well as native Linux and Xen on the older PC also shown for comparison.**

This comparison of our results to [ZPERF] leads us to believe that no simple software configuration enhancements will improve performance on our zServer, and that our figures although generated on an older model are comparable to more recent offerings from IBM. [ZPERF] also gives dbench scores for zServers with 4, 8 and 16 processors. Their results indicate that performance would be significantly better for a zServer with multiple processors. For example, [ZPERF] page 14 reports around 1000 MB/sec for a 16 CPU z900. We are also not testing all the features of the zSeries machines including high-availabilty, upgradability and manageability.

In Figure 7, we add measurements using our web server benchmark of the zServer with 1 to 16 Linux guests to the data presented in Figure 3. Xen on the Xeon server and the zServer perform similarly with the zServer performing better than Xen at 2, 4 and 16 guests, but worse at 1 and 8.

Thus, we are ready to answer our fourth question: How does a virtual machine monitor on commodity PCs compare to a virtual machine monitor on a mainframe? At least on our low-end zServer, Xen on x86 performs better for most workloads we examined. For a $2500 machine to do so well compared to a machine valued at over $200,000 is impressive!

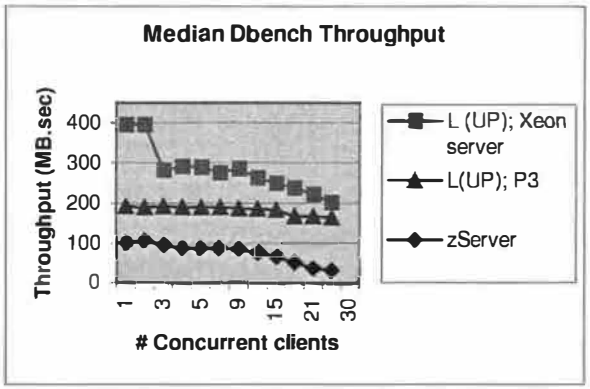



**Figure 6 Throughput reported by dbench for 1 to 24 concurrent clients for native Linux on a Xeon server, native Linux on an older PC and Linux on the zServer.**

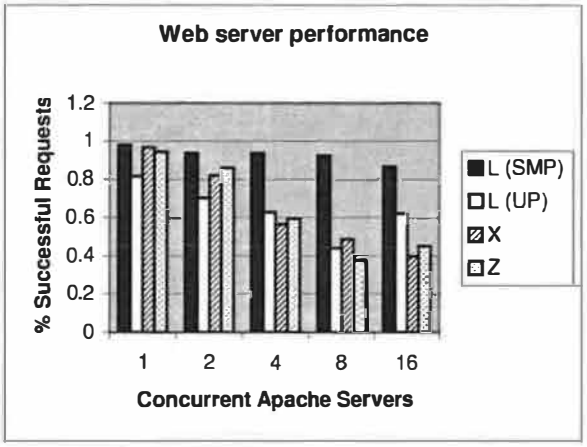



**Figure 7 Web server performance on the zServer compared to native Linux with SMP enabled, native Linux with SMP enabled and XenoLinux on a Xeon server.**

## 6. Experience With Related Research

The Xen team did a great job of facilitating repetition of their results, including releasing the code open source, producing a trial CD and responding happily to questions. Still, we were surprised to find how difficult it was to reproduce these results! It took a lot of investigation to assemble a comparable test platform and to reproduce the tests as run in [Xen03]. In the process, we ported three device drivers, wrote over a dozen testing scripts, wrote our own web server benchmark and ran hundreds of trials of each benchmark.

We make three main conclusions about repeated research. First, it is difficult enough that it should not be left as an exercise to the reader. Having another group repeat the initial results and polish some of the rough edges is important to the process of technology transfer. Second, it provides important validation of published results and can add additional insight beyond the original results. An independent third party is needed to verify the reliability of results reported, to question which tests are run, and to highlight some of the "spit and bailing wire" still present in the system. Finally, it is a great way to gain experience with research. This paper was a class project for an Advanced Operating Systems class at Clarkson University. This experience gave us a better appreciation for research in computer science than simply reading the other 30+ research papers in the class.

## 7. Conclusions

We were able to repeat the performance measurements of Xen published in "Xen and the Art of Virtualization" from SOSP-03. We find that Xen lives up to its claim of high performance virtualization of the x86 platform. We find that Xen can easily support 16 moderately loaded servers on a relatively inexpensive server class machine, but falls short of the 100 guest target they set. Xen performs well in tests on an older PC, although only a small number of guests could be supported on this platform. We find that Xen on x86 compares surprisingly well to an entry model zServer machine designed specifically for virtualization. We use this study as an example of repeated research and argue that this model of research, which is enabled by open source software, is an important step in transferring the results of computer science research into production environments.

## 8. Acknowledgements

We would like to thank the Xen group for releasing Xen as open source software. We would especially like to thank Ian Pratt, Keir Fraser and Steve Hand for answering our many emails. We'd also like to thank James Scott for help to port the driver for our SCSI controller. Thanks also to the Apache Foundation for JMeter. Thank you to our shepherd, Bart Massey for all of his detailed comments.

Many thanks to the members of the Clarkson Open Source Institute (COSI) and the Clarkson Internet Teaching Laboratory (ITL) for their patience as we tore apart lab machines to conduct our tests. Thanks also to Clarkson University and especially the Division of Mathematics and Computer Science for their support of the labs.

## 9. References

[AS3AP] C. Turbyfill, C. Orji, and D. Bitton. An ANSI SQL Standard Scalable and Portable Benchmark for Relational Database Systems. The Benchmark Handbook for Database and Transaction Processing, Chapter 4, pp 167-206.

[Denali02] A. Whitaker, M. Shaw, S. Gribble. Scale and Performance in the Denali Isolation Kernel. In Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI 2002), ACM Operating Systems Review, Winter 2002 Special Issue, pages 195-210, Boston, MA, USA, December 2002.

[Disco97] E. Bugnion, S. Devine, K. Govil, M. Rosenblum. Disco: Running Commodity Operating Systems on Scalable Multiprocessors. ACM Transactions on Computer Systems, Vol. 15, No. 4, 1997, pp. 412-447.

[FourInARow] Freebench.org Project. URL http://www.freebench.org accessed December 2003.

[Google03] S. Ghemawat, H. Gobioff, S. Leung. The Google File System. Proceedings of the 19th ACM Symposium on Operating Systems Principles, 2003.

[JMETER] The Apache Jakarta Project. Jmeter. URL http://jakarta.apache.org/jmeter/index.html accessed December 2003.

[JMETER2] A. Bommarito, Regression Testing With JMeter.. URL http://www.phpbuilder.com/columns/bommarito20030610.php3 accessed December 2003.

[JMETER3] B. Kurniawan. Using JMeter. URL http://www.onjava.com/pub/a/onjava/2003/01/15/jmeter.html accessed December 2003.

[JMETER4] K. Hansen. Load Testing your Applications with Apache JMeter. URL http://javaboutique.internet.com/tutorials/JMeter accessed December 2003.

[LKML03] K. Fraser. Post to Linux Kernel Mailing List, October 3 2003, URL http://www.ussg.iu.edu/hypermail/linux/kernel/0310.0/0550.html accessed December 2003.

[LMBENCH] lmbench, URL http://www.bitmover.com/lmbench accessed December 2003.

[OSDB] Open source database benchmark system. URL http://osdb.sourceforge.net accessed December 2003.

[POSTGRES] M. Stonebraker. The Design Of The Postgres Storage System. Proceedings of the 13th Conference on Very Large Databases, Morgan Kaufman Publishers. (Los Altos CA), Brighton UK. 1987.

[POSTGRES2] PostgreSQL Global Development Group. URL http://www.postgresql.org accessed December 2003.

[Pratt03] Ian Pratt. Personal Communication. November 2003.

[SPECINT] CPU 2000, URL http://www.specbench.org/cpu2000 accessed December 2003.

[SPECWEB] SPECWEB99, URL http://www.spec.org/web99 accessed December 2003.

[UML01] Dike, Jeff. User-mode Linux. Proceedings of the 5th Annual Linux Showcase & Conference, Oakland CA (ALS 2001). pp 3-14, 2001.

[UML00] Dike, Jeff. A User-mode Port of the Linux Kernel. Proceedings of the 4 Annual Linux Showcase & Conference (ALS 2000), page 63, 2000.

[VM370] R. Creasy IBM Journal of Research and Development. Vol. 25, Number 5. Page 483. Published 1981. The Origin of the VM/370 Time-Sharing System.

[VMWARE] Vmware, URL http://www.vmware.com accessed December 2003.

[Voxel] Voxel Dot Net, URL http://www.voxel.net accessed December 2003.

[Xen99] D. Reed, I. Pratt, P. Menage, S. Early, and N. Stratford. Xenoservers: Accounted Execution of Untrusted Code. Proceedings of the 7th Workshop on Hot Topics in Operating Systems, 1999.

[Xen03] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt and A. Warfield. Xen and the Art of Virtualization. Proceedings of the nineteenth ACM symposium on Operating systems principles, pp 164-177, Bolton Landing, NY, USA, 2003

[Xen03a] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, E. Kotsovinos, A. Madhavapeddy, R. Neugebauer, I. Pratt and A. Warfield. Xen 2002. Technical Report UCAM-CL-TR-553, January 2003.

[Xen03b] K. Fraser, S. Hand, T. Harris, I. Leslie, and I. Pratt. The Xenoserver Computing Infrastructure. Technical Report UCAM-CL-TR-552, University ofCambridge, Computer Laboratory, Jan. 2003.

[Xen03c] S. Hand, T. Harris, E. Kotsovinos, and I. Pratt. Controlling the XenoServer Open Platform, April 2003.

[Z800] IBM z800 Information, URL http://www-1.ibm.com/servers/eserver/zseries/800.html accessed December 2003.

[ZPERF] S. Thoss, Linux on zSeries Performance Update Session 9390. URL http://linuxvm.org/present/SHARE101/S9390a.pdf accessed December 2003.

# UseBSD SIG Session

# Using Globus With FreeBSD

Brooks Davis, Craig Lee
*The Aerospace Corporation*
*El Segundo, CA*
{brooks,lee}@aero.org

## Abstract

After years of development by the high performance computing (HPC) community, grid computing has hit the mainstream as one of the hottest buzzwords in computing technology today. This paper examines the issues involved in integrating FreeBSD with the Globus Toolkit, the de facto standard for grid computing. Particular attention is paid to interactions between Globus and FreeBSD's package management system. Ways in which FreeBSD could be enhanced to make it a more attractive platform for Globus are also discussed.

## 1 Introduction

Grid computing has been a hot topic in the high performance computing (HPC) community for many years now. From the earliest beginnings with the I-WAY [DeFanti] at SuperComputing 1995 to today's commercial grid products such as Oracle 10g, grid computing has entered the mainstream. Most major OS vendors including Sun, Microsoft, and IBM have grid products.

By now, most people in the computing field have heard of grid computing, but many may still ask, what is a grid? Grid computing seems to mean different things to different people. Some people associate grid computing with a specific software package such as the Globus Toolkit [Foster1] or Sun Grid Engine [SGE]. Others think of peer-to-peer systems as their prototypical grid. One early definition is from The Globus Alliance's leaders, Ian Foster and Carl Kesselman: "A computational grid is a hardware and software infrastructure that provides dependable, consistent, pervasive, and inexpensive access to high-end computational capabilities" [Foster3]. A more recent definition from Wolfgang Gentzsch is "a Grid is a hardware and software infrastructure that provides dependable, consistent, and pervasive access to resources to enable sharing of computational resources, utility computing, autonomic computing, collaboration among virtual organizations, and distributed data processing, among others" [Gentzsch].

First published in the Proceedings of the FREENIX Track: 2004 USENIX Annual Technical Conference (FREENIX '04)


The original concept of computational grids was an analogy to the electrical power grid. As the power grid provides transparent access to power from various generators run by many different operators, a computational grid provides access to distributed computer resources controlled by multiple organizations. These organizations use grid technologies to form virtual organizations which share resources to solve problems. Examples of this sharing might include computing resources such as clusters or supercomputers, sensors, instruments, or data collections.

While grid computing may have historically started in the high performance computing community, addressing the fundamental issues of *scalable deployment* for distributed information discovery, resource management, workflow management, security, fault tolerance, etc., means that grid technology will actually be applicable in many areas of the computing ecosystem. A recent compilation on grid computing [Berman] provides more information on its origins, current state, applications, and future directions.

The groundswell of interest in grid computing from both academia and industry motivated the creation of the Global Grid Forum (GGF) [GGF], a standards body modeled after the IETF. In February, 2002, work on the Open Grid Services Architecture (OGSA) and the Open Grid Services Infrastructure (OGSI) was announced at GGF. OGSI is a standard based on web services, specifically "a Grid service is a Web service that conforms to a set of conventions (interfaces and behaviors) that define how a client interacts with a Grid service" [Tuecke]. In January,

2004, work on the Web Service Resource Framework (WSRF) was announced at GlobusWorld and subsequently at GGF in March, 2004 [WSRF]. WSRF is intended to complete the convergence of grid and web services started by OGSA, whereby a service client can explicitly specify the state (resource) used by a service on any particular invocation. This essentially allows grid/web services to be considered *stateless*.

In the remainder of this paper we talk about the Globus Toolkit and its status under FreeBSD. First, we give an overview of the Toolkit. We follow this with a discussion of the state of the Toolkit under FreeBSD with detailed coverage of the "impedance mismatches" between the Toolkit and the FreeBSD Ports Collection. Next comes a discussion of ways to integrate FreeBSD and Globus more effectively followed by concluding remarks.

## 2 The Globus Toolkit

The Globus Toolkit is developed by the Globus Alliance (formerly the Globus Project) and is the de facto standard for grid computing infrastructure in high performance computing. Two versions of the Globus Toolkit are currently available with a third version on the way. The Globus Toolkit 2 (GT2) is an extended version of the original services. It is implemented in C and uses a combination of standard protocols such as FTP and LDAP in conjunction with proprietary protocols. The Globus Toolkit 3 (GT3) is a new implementation of grid services based on OGSI standards. GT3 services are implemented in a combination of C and Java. All GT2 services are included in GT3 as well as new, OGSI-compliant implementations of some services also provided by GT2. The Globus Toolkit 4 (GT4) is to be WSRF-based with a release planned for later this year.

The Globus Toolkit provides some central infrastructure and a set of orthogonal services. The most visible component of the central infrastructure is the Grid Security Infrastructure (GSI), an X.509 certificate based authentication mechanism [Foster2]. GSI provides single sign-on access to resources in multiple independent administrative domains and provides delegation of credentials via proxy certificates. On each host (and potentially on a per-service basis) a mapping is established between distinguished names and local user accounts. GSI is identical in the latest releases of GT2 and GT3 and will presumably be so in GT4.

The Globus Toolkit provides job management services via the Globus Resource Allocation Manager (GRAM) service [GRAM]. GRAM provides an interface between users or meta-schedulers and local resource managers such as Sun Grid Engine (SGE), the Portable Batch System (PBS), or a trivial fork manager. The GT2 and GT3 implementations differ in that GT3 adds a Java-based OGSI-compliant implementation. The GT4 implementations will be WSRF-compliant.

File transfer is provided by the GridFTP service. GridFTP is a set of extensions to the FTP protocol [Allcock]. These extensions include, GSI security on control and data channels, parallel transfers, partial file transfers, third-party transfers, and authenticated data channels. Both versions of the Toolkit use the C implementation.

Resource discovery is handled by the Monitoring and Discovery Service (MDS) [MDS]. The MDS is composed of two parts: (1) the Grid Resource Information Service (GRIS) which provides information about a resource, and (2) the Grid Index Information Service (GIIS) which aggregates data from GRISs to provide support for searching. In GT3, separate GRIS implementations have been removed because all OGSI services can function as their own GRIS. This should be true of the WSRF services as well.

Other services provided by the Toolkit include a Replicate Location Service and, in GT3, OGSI-compliant database services.

GT2 supports most of the major HPC platforms including AIX, HP-UX, Irix, Linux, and Solaris. It is also expected to work on most basically POSIX-like OSes. It consists of over 61 packages, some of which are open source software such as OpenSSL and OpenLDAP and some of which are products of the Globus Alliance and their contributers. The parts of GT3 that are not GT2 components are primarily Java-based Web Services.

To address the complexities of building these multiple packages on different platforms, the Grid Packaging Tools (GPT) were developed [Bletzinger]. Like the FreeBSD ports collection and the Debian package tools, GPT supports patching and building applications from source and creating binary packages for installation on multiple machines.

One unique feature of GPT is the concept of flavors. Flavors encapsulate several coarse-grained compilations options. An example of a flavor is `gcc32dbgpthr` which represents code compiled with GCC for a 32-bit architecture with debugging enabled and POSIX threads used. This is necessary in HPC environments because users often want to pick and choose among these options in order to eke every last bit of performance from their application. As a result, site administrators may find it necessary to install many different variations of parallel libraries such as those included with the Globus Toolkit. GPT facilitates this by naming all programs and libraries according to their flavor. This allows each package to be installed multiple times.

## 3 The FreeBSD Ports Collection

To understand many of the issues with integrating FreeBSD and Globus, it is necessary to understand the FreeBSD ports collection [FreeBSD1, FreeBSD2]. The ports collection is a collection of the patches and build procedures needed to download, build, and install applications. A dependency system ensures that any necessary dependences are installed before building or installing a given software package. When a port is installed, the list of files associated with it is recorded in a packing list. This list is used to remove the package or to bundle it up into a binary package. Not all ports can be packaged for legal or technical reasons, but most of the more than 10,000 ports have this capability.

Ports can hide much of the complexity involved in installation a large piece of software like the Globus Toolkit. This can significantly simplify the process of installing the software. However, porting large complex systems like the Globus Toolkit can pose significant challenges. Many of these are caused by "impedance mismatches" where the ports collection and the software being ported have different views of the way things should work.

One ports related feature that affects the Globus Toolkit is `BSDPAN`. `BSDPAN` causes standard Perl modules to be integrated with the FreeBSD packaging system. Any Perl module which uses `MakeMaker` is registered as a package. This results in the ability to upgrade modules which are available as ports via the usual methods of upgrading ports. It also allows easy removal of unneeded Perl modules via the package tools.

## 4 FreeBSD and Globus

FreeBSD is not currently being actively supported by the Globus Alliance, but GT2 builds on FreeBSD 5.x (except on amd64.) The core client software works in limited test. The fork job manager and the GridFTP daemon have been verified to work. The GRIS is at least partially functional. The current lack of information provider scripts for FreeBSD make the output of the GRIS uninformative at this point. This problem should be easily remedied by adding appropriate scripts. A longer lasting solution to this problem might be for the Toolkit to depend on a standard, external data collector such as Ganglia for some tasks. That would avoid the need to write collectors for each OS by pushing the problem off to someone else. We have not yet tested more complex uses of Globus such as SGE transfer queues over Globus [Seed].

We have created a basic port of GT2 and a supporting port of GPT. In creating these ports, we found a number of conflicts with the ports collection. Some of these are fundamental philosophical differences while others are more straightforward conflicts between Globus Toolkit defaults and the expectations of the ports collection. Many of the conflicts are actually with GPT and its assumptions rather then with the Toolkit itself.

The first few issues with GPT have to do with its use of Perl modules. In an attempt to reduce the number of dependencies required to install GPT and to keep a known baseline of module versions, the GPT installation tarball includes all the Perl modules required for operation. From the perspective of keeping the installation instructions short, this works well. Unfortunately, it comes into direct conflict with the FreeBSD package system.

One of the principles of the ports collection is that packages should not install a piece of software that is also available as a separate port. This ensures that maintenance of each piece is centralized and that OS specific patches can all be applied in one place. It also avoids multiple installations of identical files. To mitigate this, the `BSDPAN` module makes any Perl module that is installed the usual way into a package. This is nice for Perl modules that get randomly installed by hand because many of them are already ports so they can be easily upgraded by just updating the ports collection. Unfortunately, this leads to weird effects with GPT. GPT installs

standard modules in a non-standard location. On the next port upgrade cycle where there is a new version of the modules, they will be removed from there and installed in the standard location thus defeating GPT's goal of keeping the versions consistent. This probably does not matter much, but it can cause some confusion with the FreeBSD packaging tools and may result in strange double installations of some modules.

In the FreeBSD port, we have removed the code to install these standard modules and added dependencies on ports for those modules. The current solution involves patching the install scripts. Ideally, GPT should gain the ability to not install modules that already exist either based on a test for their existence or on a command line switch. In addition to installing standard modules, GPT uses a slightly modified version of the `Archive::Tar` Perl module to allow it to produce better error messages in the face of corrupt archives. This causes a number of minor problems. The problems in the previous paragraph apply, but are made worse by the fact that the next upgrade will replace this version of `Archive::Tar` with a standard one losing the modifications.

Assuming the modifications to `Archive::Tar` are necessary for GPT there are three possible solutions to this problem. One is to get the changes merged into the official release so it can be used instead of the GPT version. Given the nature of the changes, this seems unlikely. A second solution is to install the module under another name such as under `Grid::GPT::Tar` where its nonstandard behavior will remain intact and will be harmless. A third solution would be to rewrite GPT to not require these changes. The third solution would be best since that would allow the use of a system version of `Archive::Tar`.

In the port of GPT we simply use the system version of `Archive::Tar` which seems to work fine. In the context of ports, corrupt archives are not a significant issue since both the size and MD5 checksum of the archives are verified before anything is done. When built this way, GPT appears to function correctly.

One final minor issue with GPT and Perl modules is that GPT installs all Perl modules including its internal ones in a non-standard location, overriding any Perl defaults. Specifically, modules are always installed under `$GPT_LOCATION/lib/perl/`. This is fine if `$GPT_LOCATION` isn't the same prefix as the Perl install, but when it is, this is less than ideal as the system can end up with extra parallel Perl module trees. It isn't a serious problem, but it would be nice if GPT could be told to let `MakeMaker`install the modules where it wants to instead of under a hardwired location. We currently ignore this issue in the port.

Another, less fundamental, GPT issue is caused by the use of bundles. Bundles are a set of packages that generally have no external dependencies. They are used because GPT does not provide the sort of automatic dependency installation that the ports collection does. Without this sort of dependency management, installing Globus would almost certainly be too difficult if administrators had to install it from packages as they would have to install as many of 61 packages in a specific order. The problem presented by the bundles is that while they are a reasonable breakdown of the functionality of GT2, they all contain some packages such as `globus_core`. Since each file in the system must be managed by at most one port, the bundles can not be separate ports.

The simplest way to work around this is to use a single port that installs all the bundles. This is the approach we took in the current port. Another approach would be to use the ports collection's dependency management system to install the individual packages as individual ports. This should be a feasible approach. Meta ports could be used to represent the bundles easing upgrading in the face of security advisories and allowing more selective inclusion of Globus tools. One concern with this approach is dealing with packages that install binaries in `bin`, `sbin`, or `libexec` and must be configured with more than one flavor. Some method of insuring that only one flavor installs the primary executable will need to be determined or both flavors will always have to be installed. Adding a feature to the ports system that allowed ports to depend on other ports with specific build options enabled would be very helpful here.

Flavors may be another point of contention. Support for third-party compilers such as Intel's `icc` would be useful, but the ports collection does not handle this sort of thing well. Currently an option could be used to control the base flavor but if there was a desire to install multiple compiler flavors, the only option would be to create slave ports for other compilers leading to an explosion of ports. This solution would be time consuming and unsatisfying. In this

regard GPT is superior to most existing build and packaging systems including the ports collection.

On a more philosophical level the Globus Toolkit's practice of providing its own version of standard libraries such as OpenSSL and SASL is not in keeping with the policy of the ports collection that the ports or base system version of a library be used where ever possible. Using a single version of a library simplifies maintenance of FreeBSD specific patches and reduces the overall size of the system by avoiding duplication. Unfortunately, in the case of the Globus Toolkit, this is likely to be impossible, primarily due to GPT's use of flavors.

## 5 Future Work

At this point, what Globus needs most on FreeBSD is more user testing and small bits of cleanup. Most of the existing problems are not particularly serious, but may require time consuming work. For example writing GRIS data collectors is mostly simple shell scripting, but the shell scripts have somewhat strange interfaces so there is a learning curve.

In addition to these sorts of cleanup tasks, more work could be done to make FreeBSD an attractive platform for grid computing. There are two direct ways to attack this problem. One is to improve the integration of Globus with the system by making it easier to install and making it easier for base services such as SSH and FTP to take advantage of enhancements such as GSI authentication and the GridFTP extensions. Much of this can be accomplished by adding or improving existing ports. Adding support or improving existing support for GSSAPI in applications in the base system could enable those applications to use GSI authentication [RFC2743].

The second way is to enhance Globus to take advantage of unique operating system features. One idea would be to investigate adding sendfile support to GridFTP. If it were possible to do GridFTP parallel transfers via a zero-copy mechanism, performance could be significantly improved. This might require enhancing sendfile or building a new GridFTP client/server. Another option would be adding support for using Name Service Switch (NSS) to determine the mapping between local user names and x.509 distinguished names for use in GSI authentication. With the current file based approach, maintaining those mappings across a cluster of machines is a potentially time consuming task. Adding network database support via NSS is something FreeBSD (and other OSes with a NetBSD derived NSS implementation) are uniquely capable of doing because NetBSD had the foresight to include external access to NSS databases via the nsdispatch() function. Having a NSS replacement for `/etc/grid-security/grid-mapfile` could significantly decrease administrative overhead in clusters of grid enabled machines.

A third, less direct method of making FreeBSD more attractive to grid users is to work towards solving problems that are high priorities in the grid computing community. For example, many grid projects have to deal with transferring huge amounts of data over long distances. Research into improved TCP congestion control techniques such as HSTCP and XCP (and into methods to ease that research) could pay big dividends in terms of attracting these types of users [Floyd, RFC3649, Falk]. Other areas to look at include trusted computing [Watson] and storage technologies.

## 6 Conclusions

FreeBSD clients should be able to interact with any GT2-based grid as first class citizens. Most services also seem to work, but further testing is warranted and GRIS support needs some work. GT3 programs are expected to generally work, but have not yet been tested. While there are still significant rough edges, FreeBSD shows promise as a platform for Globus-based grid computing that should continue with GT4.

## References


[Allcock] W. Allcock, *GridFTP: Protocol Extensions to FTP for the Grid*, GFD-R.020, GGF. http://www.ggf.org/documents/GWD-R/GFD-R.020.pdf

[Berman] F. Berman and G. Fox and A. Hey (eds.), *Grid Computing: Making the Global Infrastructure a Reality*, Wiley, 2003.

[Bletzinger] M. Bletzinger, *A User Guide to the Grid Packaging Tools*, National Center for Super Computing Applications (NCSA) University of Illinois, 2003. http://www.gridpackagingtools.com/book.html

[DeFanti] T. DeFanti, I. Foster, M. Papka, R. Stevens, and T. Kuhfuss, *Overview of the I-WAY: Wide area visual supercomputing,* International Journal of Supercomputer Applications, 10(2):123– 130, 1996.

[Falk] A. Falk, D. Katabi, "XCP Protocol Specification", unpublished draft, February 12, 2004. http://www.isi.edu/isi-xcp/docs/xcp-spec-04.txt

[Floyd] S. Floyd, S. Ratnasamy, and S. Shenker, *Modifying TCP's Congestion Control for High Speeds*, Rough draft, May 2002. http://www.icir.org/floyd/papers/hstcp.pdf

[FreeBSD1] *The FreeBSD Handbook*, The FreeBSD Documentation Project, 2004. http://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook

[FreeBSD2] *The FreeBSD Porter's Handbook*, The FreeBSD Documentation Project, 2004. http://www.freebsd.org/doc/en_US.ISO8859-1/books/porters-handbook

[Foster1] I. Foster and C. Kesselman, *Globus: A Metacomputing Infrastructure Toolkit*, Proceedings of the Workshop on Environments and Tools for Parallel Scientific Computing, SIAM, Lyon, France, August 1996.

[Foster2] I. Foster, C. Kesselman, G. Tsudik, and S. Tuecke, *A Security Architecture for Computational Grids*, Proceedings of the Fifth Conference on Computer and Communications Security, San Francisco, CA, 1998, pp. 83–92.

[Foster3] I. Foster and C. Kesselman (eds.), *The Grid: Blueprint for a New Computing Infrastructure*, Morgan Kaufmann, San Fransisco, 1999

[Foster4] I.Foster, *What is the Grid? A Three Point Checklist*, Grid Today, 2002. http://www.gridtoday.com/02/0722/100136.html

[Gentzsch] W. Gentzsch, *Response to Ian Foster's "What is the Grid?"*, Grid Today, 2002. http://www.gridtoday.com/02/0805/100191.html

[GGF] http://www.globalgridforum.org

[GRAM] http://www-unix.globus.org/developer/resource-management.html

[MDS] http://www.globus.org/mds/

[RFC2743] J. Linn, *Generic Security Service Application Program Interface, Version 2, Update 1*, RFC2743, IETF, 2000. http://www.ietf.org/rfc/rfc2743.txt

[RFC3649] S. Floyd, *HighSpeed TCP for Large Congestion Windows*, RFC3649, IETF, 2003. http://www.ietf.org/rfc/rfc3649.txt

[Seed] T. Seed, *Transfer-queue Over Globus (TOG): How To*, EPCC, 2003. http://gridengine.sunsource.net/download/TOG/tog-howto.pdf

[SGE] http://gridengine.sunsource.net

[Tuecke] S. Tuecke, K. Czajkowski, I. Foster, J. Frey, S. Graham, C. Kesselman, T. Maquire, T. Sandholm, D. Snelling, and P. Vanderbilt, *Open Grid Services Infrastructure (OGSI) Version 1.0*, GFD-R-P.15, GGF. http://www.ggf.org/documents/GWD-R/GFD-R.015.pdf

[Watson] R. Watson, W. Morrison, C. Vance, B. Feldman, *The TrustedBSD MAC Framework: Extensible Kernel Access Control for FreeBSD 5.0*, Proceedings of USENIX '04: FREENIX, USENIX, June, 2003. http://www.trustedbsd.org/trustedbsd-usenix2003freenix.pdf.gz

[WSRF] http://www.globus.org/wsrf

# The NetBSD Update System

Alistair Crooks, The NetBSD Project

9th April 2004

## Abstract

This paper explains the needs for a binary patch and update system, and explains the background and implementation of NetBSD-update, a binary update facility for NetBSD. The implementation is then analysed, and some lessons drawn for others who may be interested in implementing their own binary update system using the NetBSD pkgsrc tools, which are available for many operating systems and environments already.

## The NetBSD Binary Update System

Unix, Linux and the BSD operating systems have traditionally been distributed in source format, and users and administrators have had a long tradition of compiling utilities and applications from source. Over time, however, vendors have moved towards a binary-only distribution mechanism, removing various parts of the system in the process, such as the C compiler, and other necessary tools. It is only over the last decade that the rise of Linux and the BSD operating systems have placed the emphasis back on source code, and even then, most versions of the operating systems are installed from a binary distribution.

This paper describes the NetBSD update system, which has been implemented on and for the NetBSD operating system `http://www.netbsd.org/` using the packaging tools from the NetBSD Packages Collection (pkgsrc) `http://www.pkgsrc.org/`.

Other vendors and operating systems have binary update facilities in place - their existence is not the driving force behind the use of pkgsrc or NetBSD - rather, this is a description of a facility which is used in NetBSD and which can be used on any other operating system to augment the standard facilities which are in place.

### Driving Forces for a Binary Patch and Update System

It is now common to find firewalls in large and small organisations, preventing malign access, and protecting the organisation from intrusion and other attacks. It would not be prudent to have a C compiler installed on such a machine - its use should be that of a gatekeeper, as a bouncer with an attitude, keeping anything suspicious out, and not allowing anyone who does manage to penetrate the defences to use any tools to break further into the infrastructure.

In addition to these instances, it is very unusual to find users (as opposed to administrators, or people who work in the industry) who would know what to do with a source distribution. Email to various mailing lists proves this point - to the majority of users out there, the computer is a tool, not a thing of beauty.

It is also common to find vulnerabilities in operating systems, libraries, and utilities which have already been deployed. To fill such holes, a patching system needs to be used - the vulnerable code is replaced by code which is not vulnerable. This must be done by means of updating the binaries.

Embedded systems must also be examined in light of the vulnerabilities found in MTAs, network time servers, IP filtering software, operating system software, and anything else which is included as part of

the embedded system, and which can be used to facilitate an attack on an organisation or individual - the whole infrastructure is only as strong as the weakest link in the defences, and one breach renders the rest of those defences useless.

It is not feasible to expect that all the vulnerabilities can be found and protected ahead of time - new ones are being found every day - and so there must be a mechanism available by which new binaries can be used to overwrite vulnerable ones. This paper examines all the issues involved in the design and deployment of such a system.

It can take some time for an advisory to be received, investigated, researched, fixed, and then publicised - the binary update facility can be viewed as a more reliable form of communication of the exploit or vulnerability than reading the Bugtraq or Full Disclosure mailing lists.

Some vendors wish users and administrators to pay for the added service which a binary update facility provides. Whilst the NetBSD binary update system is not intended to prevent vendors charging for this service, some people are unhappy and unwilling to pay for a binary update system.

## Related Systems

The one immediate piece of software to which everyone refers when talking about binary updates is the Windows Update Facility `http://v4.windowsupdate.microsoft.com/en/default.asp`. It has three separate facilities - update check (for new upgrades), update download, and update installation, and Windows XP computers can be set to perform the check, the check and download, and all three parts automatically. Windows Update is based around a web front end - all the used has to do is to specify (the first time Windows Update is run) how they would like their system to check for updates - and everything is very easy from an end-user point of view.

Debian Linux `http://security.debian.org/` has a security update feature which uses its apt system to provide binary patches for Debian Linux systems.

The FreeBSD project `http://www.freebsd.org/`are working on their own binary update system.

The RedHat network `http://www.redhat.com/` provides a commercial service for RedHat Linux systems.

Sun Microsystems `http://www.sun.com/` provides a number of binary updates via its SunSolve facility for its Solaris operating environment.

The NetBSD Packages Collection `http://www.pkgsrc.org/` has an "audit-packages" package which is used to notify the user when packages which are installed on the host system have been found to have security exploits or vulnerabilities. A central list is maintained by the security-officer and other developers, and stored on one of the NetBSD project servers. The user is encouraged to run the "download-vulnerability-list" as part of the periodical jobs on a host system by means of the crontab(8) facility. This script will download the list of vulnerabilities from the central server. Output from the "download-vulnerability-list" script looks is reproduced in the Figure download-vulnerability-file output.

A separate script, called "audit-packages" simply runs through each of the entries in the list of vulnerabilities, and checks against each of the vulnerabilities for an installed package on the system with a version number that is vulnerable. If there is, then a warning message will be printed, with relevant information, as shown in Figure audit-packages.

The vulnerabilities file is split into 3 columns. Each line of the file describes a new vulnerability. Comments start with the '#' character. The first column is the vulnerable package versions; the second column is the type of vulnerability, and the final column is a URL related to the vulnerability.

Some typical entries from the vulnerabilities list are shown in the Figure pkg-vulnerabilities file.

The audit-packages package works very effectively in practice, having ironed out a few loose ends in its implementation. From the start, the vulnerability list has been advertised that it will only grow - old vulnerabilities will be retained "just in case". This has proved difficult to implement - occasionally URLs will be found to be out of date, or better ones identified. Once a user with a shorter username fixed two transposed characters in a comment, thereby causing the size of the vulnerabilities list to decrease. In time, it was found better to have an embedded SHA1 di-

gest, which is used to ascertain correct tranmission of the vulnerabilities list, and that no truncation has occurred in transit, as well as ensuring that the file has not been modified in any way.

NetBSD also has fine-grained "system packages", which can be used as an alternative to the traditional BSD sets method of installing or upgrading systems. System packages are usually made up of three sub-packages:

- the binaries,
- the manual pages,
- and any example or support files.

The NetBSD Update System does not use any of the facilities of system packages - on non-NetBSD systems, it is likely that system packages would clash with traditional methods of installing and updating systems, and so a separate mechanism was used to implement the NetBSD Update System.

## The Design of the NetBSD Update System

The audit-packages package, and the experience gained from it, proved to be a major influence in the design of the NetBSD update system. By using a single list of vulnerable packages which was downloaded, significant useful information could be retrieved very simply and in a relatively secure manner.

The utility and ease-of-use of the Windows Update system proved to be the inspiration for the actual operation of the system. In addition to the ease of use, the three stages of protection in the Microsoft utility were considered to be the correct approach. In some countries, users must opt-in to any service which holds information, and the binary update facility was designed with this in mind.

Security and integrity were two more considerations - security in the transit of information which could be potentially damaging to a system if the update were modified in any way during transit. For this reason, digital signatures of the update itself are considered absolutely essential.

The author has some experience with the packaging tools as used in the NetBSD packages collection, and certain facilities used in these tools would be needed in the binary update facility itself. Using version numbers which could be compared rationally with each other, and using the tools themselves to add, backup and delete files, directories and other file system entries are essential in providing a useful utility which does not try to do any more than it needs to do.

## Implementation of the NetBSD Update System

The implementation of the NetBSD update system is done as in the Windows and Debian updates - the binary updates can be set to inform of new updates, to inform and download, and to inform, download and install the updates automatically.

The NetBSD packages collection tools are used to perform the binary update work - these tools have been enhanced from the original tools to add such features as "Dewey decimal" number comparison, digitally-signed package checks and addition, and package replacement in place. There is also a facility for a binary package to be specified by URL on the command line, downloaded automatically, and added via pkg_add(1), but that facility is not used by the NetBSD binary update system - instead, ftp(1) isused to download the binary package, and pkg_add(1) is used in "local" mode. This is because we prefer the binary packages which are being downloaded to be protected by a digital signature, and that mode of binary package addition is, unfortunately, not yet available when specifying a URL on the command line.

For a long time, the NetBSD pkgsrc system has included a package called "pkg_install" - it consists of the latest version of the pkg_install tools. This was derived from the original FreeBSD pkg_install(1) tools (and is also included in the base NetBSD operating system), but has been considerably enhanced, including specifically such such features as

- the recognition of packages by means of relative version numbers

- the addition of a callout to gpg and pgp to authenticate the provenance of a binary package by means of a digital signature related to the binary package. This had to be implemented as a callout as gpg is distributed under the GPL, and it is not desirable to have any part of this system distributed under a more restrictive or onerous licence than the BSD licence. The implication of this is obvious - the BSD operating systems need a BSD-licensed utility for privacy, compatible with gpg and pgp. One example of an addition of a signed binary package is shown in Figure Addition of a signed binary package.

The recognition of packages by comparitive version numbers is essential for any binary update system. For example, it is imperative that we can tell whether an installed package is vulnerable to an exploit. There are a number of ways this could be done

- circulate digests of files that are vulnerable, but that is not practical in an environment such as the BSD operating systems where most packages are built by the user, and binary packages, whilst still being used by many people, are not as popular as the "build it for myself" attitude. The author suffers from this affliction as well, and does not consider it a weakness, more the signs of knowing exactly what is on any given system at any one time.
- by hardcoding the package names which are vulnerable, but this does not scale well, or would even be practical
- by using a system package name and version number (but, as we have already seen, this is not portable to operating systems other than NetBSD)

The NetBSD packages collection has also long had a number of"meta-packages" - these are packages which include no files or directories of their own, but which require pre-requisites of other packages, and thereby ensure that all of the parts can be manipulated as one whole package. Examples of meta-packages are:

- edesktop
- gnome
- gnome2
- gnustep
- kde3
- netbsd-www
- etc

## The First Implementation

As it was first implemented, the NetBSD binary update system was actually a fairly simple shell script which used off-the-shelf tools to perform its work. Some thought had been placed into making it into a standalone compiled program, since that is the vehicle which would most readily benefit embedded systems vendors, and that is left as future work.

The system itself consisted of two parts:

- an $opsys-$osversion-$architecture-update meta-package
- the individual updates in the form of binary packages

The netbsd-update facility first checked the directory on the NetBSD ftp server to see if there was a more recent meta-package than the one currently installed on the host machine. If there was, this was notified to the user.

Then, when the time came to download the new updates, the new meta-package was downloaded, as well as any binary packages which were necessary. Whilst it was usual for an update to include one binary package for that architecture and version, more may be needed. It's also possible that a binary package may itself have been updated, in which case the newer version of that was downloaded, along with the updated meta-package.

The operating system version and architecture are used in the package name to ensure that the correct binaries are downloaded (the architecture), and that the binary package was linked with the correct system libraries (operating system version).

When the time came to install the binary updates, the normal pkg_add(1) tool was used, working from a local copy of the package. The "-s" switch was used to ensure that the digital signature (in the form of a separate ASCII-armoured file) for the binary package verified the update's authenticity (much as a signed RPM package is used in Linux).

## An overview of the NetBSD Update Facility

Building on the prototype building blocks outlined above, the NetBSD Update facility has to perform the following steps:

1. ascertains the current operating system name and version
2. downloads a list of the binary packages available for the operating system and version of the operating system
3. evaluate whether any binary packages are needed - the same process which is undertaken by audit-packages takes place here. The list of available binary packages is processed, using pkg_info(1) to work out whether the binary package is installed on the system or if it needs to be.
4. if the user has specified that they just want to be informed of the existence of an update (which is not recommended), then this takes place by means of mail to root, and no further actions take place.
5. the necessary available packages are downloaded from the ftp server, along with their digital signatures.
6. if the user has specified that they just want the available binary packages downloaded, then they are informed by mail (to root) that this has happened, and no further actions take place
7. the files which will be overwritten (if any) by extracting files from the available binary packages are preserved by means of a simple tar(1) command, using the file list from the available binary packages, and made into a binary package of its own. This also facilitates rollback, should an unsuccessful binary package addition take place, or the new behaviour with the update in place is not working as intended
8. the binary package is fed to the pkg_add(1) utility, which first checks the digital signature on the binary package and verifieds it It is possible to specify that any signatories which match a regular expression will be allowed - for example "security-officer@netbsd\.org", or ".*@pkgsrc\.org"
9. the binary package is added to the system by means of pkg_add(1).

Consequences of these steps:

- the user is able to specify in advance whether they want the binary package notification, automatic download, or applied to the system. This is the same system as the Windows Update facility, and has been found by numerous people from small users to large enterprises to work very well.
- digital signatures protect the user by assuring them of the provenance of the binary package. Without this assurance, it would be risky or downright dangerous to apply third-party binary packages to any system, far less one which may be in production use.
- the ability to rollback from an update is simply the deletion of the downloaded binary package, and the re-application of the preserved binary package.
- people must be running a GENERIC kernel for the update system to be of any use. This is not really a problem in reality, since very few people tailor or customise their kernel configurations (usually, the only people that do that are developers)

## Feedback from Initial Usage

- The user interface was originally clunky, and needed to be made smoother

- Problems with Firewall-1 and interaction with NetBSD's ftp server mean that http may well be a better vehicle for binary update downloading
- The digital signature and automatic updates do not co-exist very well together. It is anticipated that the pkg_install tools will be modified to use a configuration file for trusted signatures.
- Operating systems which are not NetBSD were not catered for very well. Having said that, there was nothing to preclude their use on other operating systems, although duplication of updates from Sunsolve, for example, are not useful in themselves; what is useful, however, is a coherent set of packages for a given release of Solaris, for relevant architectures and machines. It has been the author's experience in the past, when managing networks of Solaris machines, that Sun provide the updates, but do not group them together particularly well, and that the numerical nature of the updates names are not the easiest to remember or inform others.
- The quality of the binary updates is only as good as the port-masters for NetBSD who have to build the binary packages.
- Making meta packages for each individual architecture is onerous and time-consuming. Some knowledge of package creation is needed.

## Iterative Development

Learning from the experience gained from the prototype implementation, it was decided that some changes would benefit everyone:

- rather than having a separate meta-package for each operating system, and operating system version, and architecture, a single vulnerabilities file was used, mirroring the vulnerabilities file in the "audit-packages" package
- the vulnerabilities file can also use an embedded digest, as a further check that correct transmission has taken place - although this is not as useful or protective as:
- a digital signature of the vulnerabilities file can be added in a trivial manner, to protect the integrity of the information
- using a simple vulnerabilities file means that there is no confusion of different files per architecture or operating system, no duplication of information

The refined binary update facility now performs the following steps:

- download the vulnerabilities file from the NetBSD ftp server
- ensure its integrity and provenance by calling out to gpg to verify the digital signature
- if there is a vulnerability on the host system, this will be flagged by a down-level update package (from a previous vulnerability fix), or by the update package not existing on the host machine.
- if the update package is to be downloaded, use ftp(1) to download the binary package from the NetBSD ftp server. At the same time, the digital signature will be downloaded. Typically, they will both be downloaded together as one entity, to try to minimise any spoofing attacks, or man in the middle attacks, since ftp is hardly the most secure of protocols
- if the update package is to be installed automatically, if there are any files to be preserved (so that rollback can take place), then the package tools are used to create a binary package of the preserved files. The list of files to be preserved will be included with the update package itself as part of its meta data.
- if the update package is to be installed, use pkg_add(1) with its "-s gpg" argument to verify the package's integrity, and to add it
- the root user will be informed of all of the above steps by email. It is believed by the author that it is better to err on the side of too much information, especially where far-reaching changes to

the system could be taking place. Update packages can have their information displayed using the normal pkg_info(1) interface.

## Conclusions

The binary update system is a useful piece of work, which has shown positive benefits.

- The ability to provide better security warnings and recovery is an enormous benefit
- even if a fix is not available, the binary update system can be used to publicise an exploit in the wild, thereby making the administrator's job easier (assuming that information about the exploit is available). Even being aware that a vulnerability has been found in a widely-used piece of software is a valuable service
- the use of digital signatures to protect and disseminate the information has proved to be of immense benefit

There are certainly areas for development and improvement, but the cross-platform and cross-environment nature of the NetBSD packages collection can aid other operating systems and environments as well as simply the BSD operating systems.

## Future Work

The NetBSD packages collection (pkgsrc) runs on the following operating systems:

- AIX
- BSDOS
- Darwin (Mac OS X)
- FreeBSD
- Interix (Microsoft's Services for Unix)
- IRIX
- Linux
- NetBSD
- OpenBSD
- SunOS (Solaris)

whilst more - HP/UX, the Hurd, Digital Unix - are planned. We would like to extend the binary update facility to run on as many of those operating systems as possible, as the benefits are measurable. It is certainly the case that duplication of work would be inadvisable, but the binary update facility offers a real benefit to end users.

It would be beneficial to investigate using a protocol that is more secure than ftp to download the vulnerabilities file, update packages, and digital signatures. Some form of anonymous secsh could be useful for this project.

A BSD-licensed gpg utility, perhaps based on openssl, would mean that callouts to gpg could be avoided, and that the whole binary update system would be licensed with a less restrictive licence than the GPL.

## References


1. The NetBSD Operating System - http://www.netbsd.org/
2. The NetBSD Packages Collection - http://www.pkgsrc.org/
3. The Microsoft Windows Update Facility - http://v4.windowsupdate.microsoft.com/en/default.
4. Debian Linux Security Update - http://security.debian.org/
5. The FreeBSD Project - http://www.freebsd.org/
6. The Red Hat Network - http://www.redhat.com/
7. Sun's SunSolve database - http://sunsolve.sun.com/

Figure 1: download-vulnerability-file output

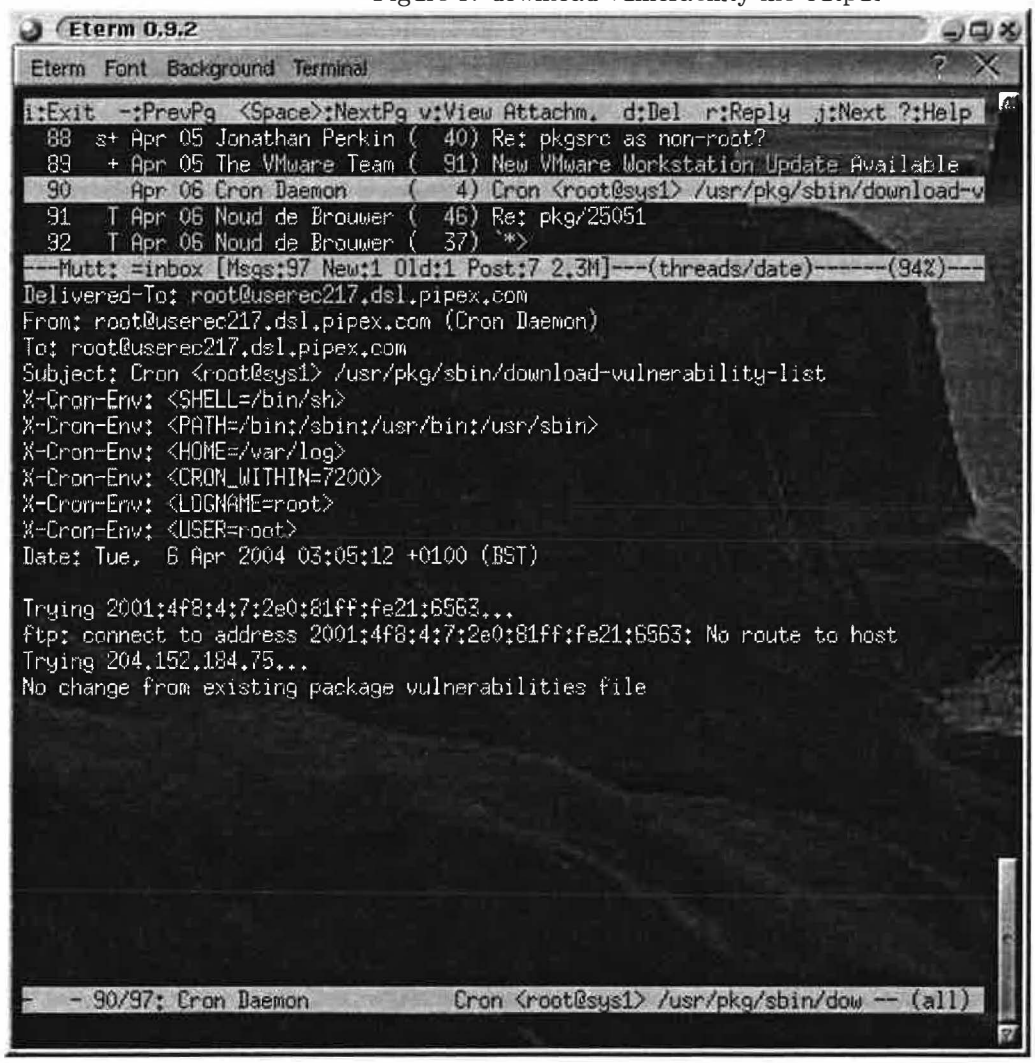


Figure 2: audit-packages

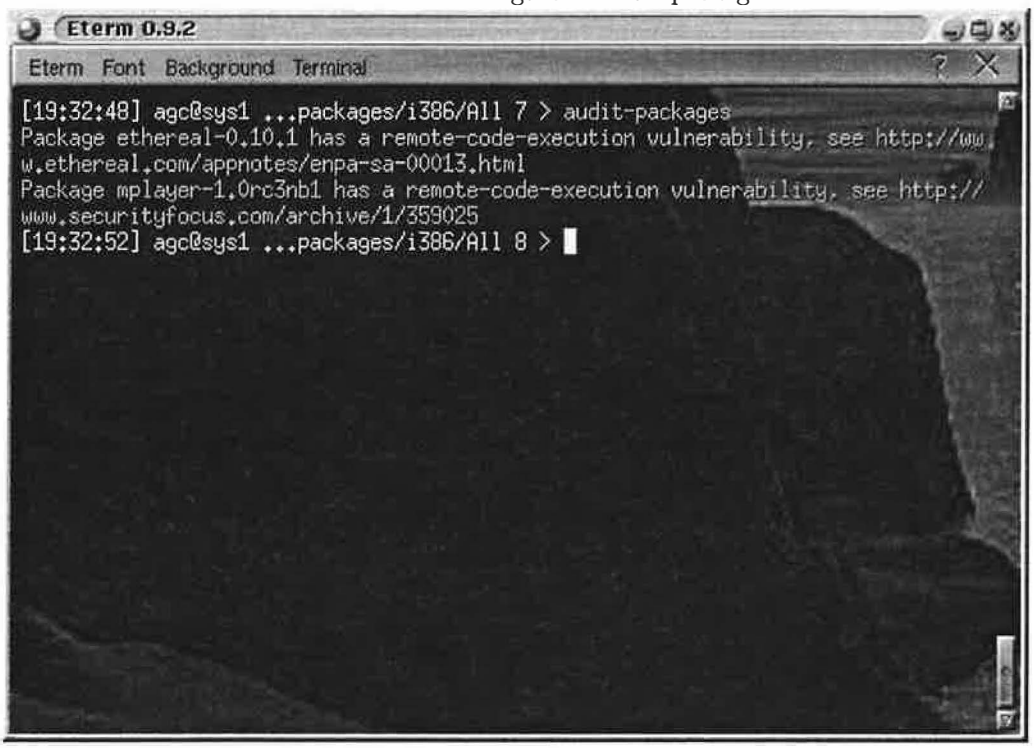


Figure 3: pkg-vulnerabilities file

Figure 4: Addition of a signed binary package

# A Software Approach to Distributing Requests for DNS Service using GNU Zebra, ISC BIND 9 and FreeBSD *

*Joe Abley*
Internet Systems Consortium, Inc.
950 Charter Street, Redwood City, CA 94063, USA
jabley@isc.org

## Abstract

This paper describes an approach for deploying authoritative name servers using a cluster of hosts, across which the load of client requests is distributed. DNS services deployed in this fashion enjoy high availability and are also able to scale to increasing request loads in a straightforward manner.

The approach described here does not employ any custom load-balancing appliances (e.g. devices commonly marketed as as "layer-four switches," "content switches" or "load-balancers"); instead the individual members of the cluster announce a service address to one or more gateway routers by participating in routing protocols to provide an intra-cluster anycast architecture.

The F Root Name Server is deployed using clusters built in this fashion, using FreeBSD [1], GNU Zebra [2] and ISC BIND 9 [3].

## 1 Design Goals

This paper describes a alternative to the deployment of individual authoritative DNS [4] servers which seeks to achieve the following:

1. Reliability: The DNS service should be highly available, and should hence survive both unplanned single-point events such as hardware failures, and also planned maintenance which might cause individual components to become unavailable.

2. Integrity: Software problems on individual hosts in the cluster should result in the corresponding hosts being automatically removed from service, so their impact on the service being provided is minimised.

3. Scalability: The DNS service should be capable of scaling to handle very high request loads without requiring very high performance on individual hosts. In combination with the reliability requirement above, upgrades to allow increasing request loads to be handled should be possible without making the service unavailable.

The architecture described in this paper could also be applied to other services. DNS, however, is ideally suited to this approach by virtue of its stateless (or stateful but short-lived) transactions; other protocols with different characteristics may not fare as well. See Section 6.4 for more discussion.

The use of this technique for load-balancing authoritative DNS service has been widely tested in the field by ISC in the deployment of the F root name server.

## 2 General Approach

One or more routers are connected to a common, multi-access network such as a single VLAN on an Ethernet switch. Two or more hosts are also connected to this common subnet (see Figure 1).

Individual interfaces on hosts and routers are configured with globally-unique addresses, such that each interface on each component can be addressed unambiguously by other devices on the Internet. These unicast addresses are used for management and other non-service traffic.

A service is associated with an IPv4 or IPv6 service address. This address is different to any of the addresses configured on any of the router or host interfaces described above.

Each host is configured with a dedicated software loopback interface, on which just the service address is configured.

The routers and hosts are all configured to participate in an OSPF [6] backbone area, through which hosts signal reachability of the service loopback address using OSPF Link State Advertisements (LSAs).

Requests from clients are sent to the service address. Each request will be delivered to a single host for processing, and hence the DNS software on each host should be configured to listen for requests on that service address. Responses from

*This paper contains material originally published in ISC-TN-2004-1 [5]. That document contains configuration examples which have been omitted from this paper in the interests of brevity.

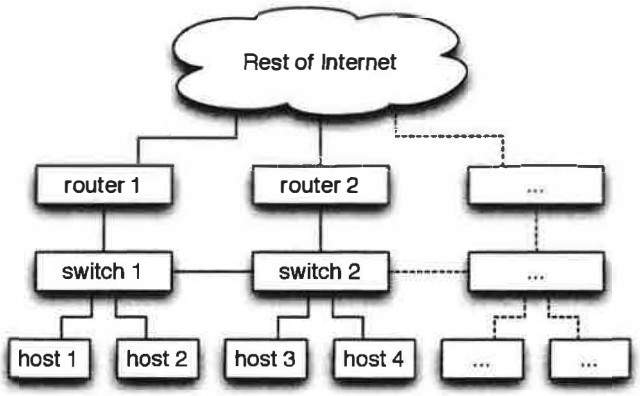



Figure 1: Components of the Anycast Cluster

the host being sent back to a client are generated with the source address set to the service address.

Non-service traffic sent from outside the cluster to individual hosts (e.g. management traffic) is directed to the globally-unique, unicast address of each host. Corresponding response traffic originated from the host is sourced from that unicast address, and not the service address.

Each router provides routes to the rest of the Internet, probably including a gateway of last resort.

Each router now enjoys multiple paths to the service address, learnt through OSPF. The routers are configured to route packets destined for the service address using some appropriate heuristic to obtain the desired load-balancing (see Section 3.2).

# 3 Routing Details

## 3.1 General

In the context of OSPF, each host is a router advertising availability of a link connected to a distant common subnet which covers the service address. The SPF algorithm when run on the two gateway routers (labelled "router 1," "router 2," etc. in Figure 1) yields multiple, equal-cost candidate routes for the service address; these are described as Equal Cost Multi-Path (ECMP) routes in the OSPF specification.

The host route for the service address is anycast within the routing system of the cluster. The service address may appear to be a unicast system as viewed from the Internet as a whole, or the cluster may be one of many providing the same service as part of a distributed, Internet-wide anycast deployment [7].

## 3.2 Gateway Routers

It is a requirement that the gateway routers used in the cluster are able to make use of all available ECMP routes. Although supported by OSPF, availability of ECMP support may be limited by the routing architecture of the system on which OSPF is implemented: for example, some operating systems cannot accommodate more than one route to a single destination.

### 3.2.1 Stateless Transactions

For DNS requests carried over UDP [8] with no fragmentation, an entire transaction consists of a single-packet request followed by a single-packet response. The protocol is stateless and it is acceptable for subsequent packets sent to the service address by the same client to be delivered to different hosts. Any route selection algorithm on the gateway routers will provide an distribution of request traffic which allows DNS transactions to proceed.

### 3.2.2 Stateful Transactions

**Flow Hashing** DNS transactions which require multiple packets to be exchanged between client and server cannot accommodate successive packets being delivered to different hosts. This is the case, for example, for DNS transactions which are performed over TCP [9], since state is maintained on an individual host between packets for a single transaction.

For DNS transactions carried over TCP it is necessary to associate a single route from the set of candidate ECMP routes with all packets associated with a single transaction (a "flow"). Cisco routers using Cisco Express Forwarding (CEF) [10] are able to associate a hash of (source, destination) internet- and transport-layer addresses with a single route, which satisfies this requirement.

CEF's route selection algorithm is stateless and deterministic for a stable set of ECMP routes. In general, however, a change in the number or ordering of those routes may cause the route selected for a particular (source, destination) hash to change. This fragility should be considered when gauging whether this load distribution approach is appropriate to particular protocols. See Section 6 for further discussion.

Juniper routers can be configured with `load-balance per-packet` which, on routers with the Internet Processor II ASIC, provides similar forwarding behaviour to that described for CEF.

Load distribution between hosts using flow-hashing forwarding algorithms will tend to be uneven in terms of traffic presented. The degenerate, illustrative case of this is a single host sending a stream of requests to the service address from a consistent source port: requests will always be answered by a single host, since the (source, destination) hash will always select a single route. See also Section 6.1.

**Upstream Router Considerations** Different gateway routers will, in general, map the same (source, destination) hashes to different candidate routes, since the ordering of ECMP routes in each router will be different. In order to ensure that packets from a single flow are routed to a single host for processing it is necessary that all the packets enter the cluster via the same gateway router.

This imposes the requirement on upstream routers that the route to the service address be stable for a single flow, which can be accommodated, for example, by having those routers use flow hashing forwarding algorithms with ECMP routes, or routing protocols which explicitly deny ECMP such as standard BGP [11] without multipath extensions.

### 3.3 Hosts

Hosts do not share the forwarding requirements described in Section 3.2; datagrams sourced from the hosts are unambiguously addressed to the unicast addresses of clients, and any equal-cost route diversity in the path back to those clients will inevitably converge on a single device.

The availability of the DNS service on a particular host is signalled to the gateway routers by issuing an LSA through which the service address is reachable. Correspondingly, the non-availability of the DNS service is signalled by issuing an LSA which withdraws the route, as if the link between the "host" router and the service address had been severed.

The straightforward requirements for the host's routing ability are easily satisfied by unix-based OSPF implementations such as the one included in GNU Zebra.

## 4 Host Operating System Considerations

The operating systems used on the hosts support cloneable Loopback interfaces, and a dedicated software loopback address is created and configured with the service address. The availability or non-availability of the DNS service is then signalled to the OSPF process on the host by simply raising or lowering the interface (e.g. `ifconfig lo1 up`, `ifconfig lo1 down`).

Nodes of the F root name server are hosted on FreeBSD 4 (with support for cloneable loopback interfaces added) and FreeBSD 5 (which supports cloneable loopback interfaces as released).

## 5 Name Server Considerations

### 5.1 General Configuration

The DNS software running on the host is configured to:

1. Bind to the service address, configured on a loopback address on the local host, and listen for client requests;
2. Bind to the unicast address of the host's interface on the cluster subnet at all other times (e.g. to perform zone transfers).

### 5.2 Zone Transfers

Special consideration may be required to accommodate zone transfers from master services which provide access control based on source address.

In the case where a master server insists that a slave server's service address be used to source a zone transfer request, it will frequently be the case that traffic from the master server will be delivered to a different host from the host which originated the request. Zone transfers in this case will time out.

In an N-host cluster (assuming random distribution of flow hashes on the gateway routers) one zone transfer request in N can be expected to succeed from any host. To help ensure that the transferred data is available quickly to all hosts in the cluster, each host can be configured to attempt zone transfers from the master server and also from each other.

Some slave name servers may be configured to perform zone transfers from a mixture of master servers which accept zone transfer requests sourced only from the service address, and others which accept zone transfer requests sourced from individual hosts' unicast addresses. To facilitate this ISC BIND 8 and ISC BIND 9 will attempt zone transfers from their configured `transfer-source` address first, and will retry using an unbound socket if the first attempt fails. Using an unbound socket has the effect in this case of sourcing the zone transfer request from the local unicast address.

Where zone transfers are authenticated using methods which do not rely on source address checking (e.g. using TSIG [12]), or where zone transfers are not authenticated, zone transfer requests may be sourced from hosts' unicast addresses and the concerns described here are avoided.

### 5.3 Self-Consistency

ISC BIND 9 is designed in such a way that anomalous run-time conditions which might lead to defective behaviour cause the named process to terminate. This facilitates a straightforward automatic control mechanism to allow the advertisement of the service to be tightly coupled to the availability of the DNS software, using a wrapper script.

### 5.4 Troubleshooting

For troubleshooting purposes, it is sometimes useful to be able to identify the individual host in a cluster which is servicing a particular client. ISC BIND 9 will answer a query for a TXT record in the CHAOS class for `HOSTNAME.BIND` with RDATA coresponding to the local hostname (see Figure 2).

```
$ dig +short @F.ROOT-SERVERS.NET \
>   HOSTNAME.BIND CH TXT
"sfo2a.f.root-servers.org"
$
```

Figure 2: Troubleshooting with `HOSTNAME.BIND`

Support for the `HOSTNAME.BIND` query is included in ISC BIND 8 (all versions) and ISC BIND 9 (from version 9.3).

## 6 Limitations

### 6.1 Load Balancing

The load distribution scheme is limited by the ECMP route selection algorithm used in the gateway routers. More sophisticated load-balancing algorithms are supported by dedicated load-balancing appliances (e.g. "least loaded server," "least recently used server").

### 6.2 Service Monitoring

The integrity of the service is sustained by rigourous checks on configuration files prior to the service starting, coupled with the self-consistency checks in ISC BIND 9 which cause the process to exit if they fail. External consistency checks are possible based on transactions against individual hosts' unicast address, and against the service address in general. Testing transactions directed at the service address on specific hosts is not straightforward due to the anycast routing of the service address.

### 6.3 Host Identification

Although the `HOSTNAME.BIND` lookup illustrated in Figure 2 provides some degree of diagnostic support to troubleshooting, it is not possible in general to determine the precise host which served a response to a particular DNS query without resorting to packet capture or expensive query logging on every host which is able to provide service to a client. This is an issue common to all load balancing techniques, and not just the one described in this paper.

At the time of writing work is underway to extend EDNS to provide a method to identify individual hosts from transaction response data.

### 6.4 Applicability to Other Protocols

DNS request and response traffic has the characteristic that transactions tend to be short-lived, and are executed rapidly. Other protocols whose transactions are longer lived may suffer from changing flow hash results as the ECMP route set changes; this might happen, for example, following LSAs sent from a host whose DNS service is taken down for maintenance, due to a failure in an individual host or because the cluster is being enlarged.

The deployment of the F root name server using the technique described in this paper provides data to suggest that this technique is effective for DNS service; those wishing to deploy protocols whose transactions are substantially different in nature to DNS are advised to test thoroughly.

## 7 Acknowledgements

Much of the material in this technical note is based on work done by Paul Vixie, Stephen Stuart and Peter Losher in providing service for the F root name server.

## References

[1] The FreeBSD Project, `http://www.freebsd.org/`.

[2] GNU Zebra, `http://www.zebra.org/`.

[3] ISC BIND 9, `http://www.isc.org/sw/BIND/`.

[4] P. Mockapetris, *Domain Names – Implementation and Specification*, STD 13, RFC 1035 (1987).

[5] J. Abley, *A Software Approach to Distributing Requests for DNS Service using GNU Zebra, ISC BIND 9 and FreeBSD*, ISC Technical Note Series, ISC-TN-2004-1, `http://www.isc.org/pubs/tn/isc-tn-2004-1.html` (2004).

[6] J. Moy, *OSPF Version 2*, STD 54, RFC 2328 (1998).

[7] J. Abley, *Hierarchical Anycast for Global Service Distribution*, ISC Technical Note Series, ISC-TN-2003-1, `http://www.isc.org/pubs/tn/isc-tn-2003-1.html` (2003).

[8] J. Postel, *User Datagram Protocol*, STD 6, RFC 768 (1980).

[9] J. Postel, *Transmission Control Protocol*, STD 7, RFC 793 (1981).

[10] Cisco Systems, "Cisco Express Forwarding (CEF)", `http://www.cisco.com/warp/public/cc/pd/iosw/iore/tech/cef_wp.html` (2002).

[11] Y. Rekhter and T. Li, *A Border Gateway Protocol 4 (BGP-4)*, RFC 1771 (1995).

[12] P. Vixie, O. Gudmundsson, D. Eastlake and B. Wellington, *Secret Key Transaction Authentication for DNS (TSIG)*, RFC 2845 (2000).

# UseLinux SIG Session

# The FlightGear Flight Simulator

Alexander R. Perry
*PAMurray, San Diego, CA*
*alex.perry@flightgear.org*
*http://www.flightgear.org/*

## Abstract

The open source flight simulator *FlightGear* is developed from contributions by many talented people around the world. The main focus is a desire to 'do things right' and to minimize short cuts. FlightGear has become more configurable and flexible in recent years making for a huge improvement in the user's overall experience. This overview discusses the project, recent advances, some of the new opportunities and newer applications.

## Introduction

The open source flight simulator *FlightGear* has come a long way since first being showcased at LinuxWorld in San Jose.

In April 1996, David Murr proposed a new flight simulator developed by volunteers over the Internet. This flight simulator was to be distributed free of charge via the Internet and similar networks. Curt Olson made a multi-platform release of FlightGear[1] in July 1997.

Since then, it has expanded beyond flight aerodynamics by improving graphics, adding a shaded sky with sun, moon and stars correctly drawn, automatically generated world-wide scenery, clouds and fog, head up display and instrument panel, electronic navigation systems, airports and runways, network play, and much more.

Recent changes to the simulator have simplified the customization of those features by the user, as discussed in more detail below. Instead of being in the source code, the configuration data is now specified on the command line and/or accessible using menu items and/or loaded from simple files.

## Simulator Portability

FlightGear aims to be portable across many different processors and operating systems, as well as scale upwards from commodity computers. The source has to be clean with respect to address width and endianness, two issues which inconvenience most open source projects, in order to to run on Intel x86, AMD64, PowerPC and Sparc processors.

In addition to running the simulation of the aircraft in real time, the application must also use whatever peripherals are available to deliver an immersive cockpit environment to the aircraft pilot. Those peripherals, such as sound through speakers, are accessed through operating system services whose implementation may be equivalent, yet very different, under the various operating systems. FlightGear currently supports Windows, FreeBSD, Linux, Solaris, MacOS, Irix and OS-X.

For those services which are common across most video games, the independent project *PLIB* offers a simple API that acts as a **P**ortable **Lib**rary[2]. Compared to Windows, MacOS and the Unix's, the various distributions and releases of Linux-based operating systems are very similar. There are important differences, most of which cause problems when trying to build and test *PLIB*, so these rarely impact FlightGear directly. Once the facilities required by video games are available through the Portable Library, the remainder of the code acts as a conventional application.

Any Linux user can download the source, compile it and safely expect it to run. FlightGear, and other applications with extensive 3D visual effects as in figure 1, may use different libraries under the same Linux distribution so that the binary may not be portable between computers. Some hardware only has accelerated 3D under XFree86 version 3, other hardware requires version 4, and their GLX APIs differ. This is being addressed by the *Linux OpenGL Application Binary Interface* (ABI)[4] but continues to be a source of frustration for new would-be users.

Once the issues associated with running the simulation program on a specific computer are resolved, everything else is portable. The configuration information can be freely transferred across processors and operating systems. Each installation can benefit from the broad range of scenery, aircraft models, scenarios and other adaptations that have been made available by any user.


Figure 1: Pilot's view from a Cessna 172 making a poor approach for landing at San Francisco International Airport. The "Time of Day" pop up menu is partially visible, with seven quick selects and an additional button for typing in the time. Seven consecutive screen dumps, from a wide screen display, have been combined to demonstrate the simulator's lighting model

## Simulator Structure

Unlike proprietary commercial PC flight simulators, the Open Source nature of the FlightGear project permits modification and enhancement. Since the project wishes to encourage its user community to embrace, extend and improve the simulation, the FlightGear source offers a flexible framework.

The FlightGear source tree is only one level deep, except that all the flight data models are each in their own subdirectory located under the FDM directory. Each directory contains a few header files that expose its object definitions. Other source files refer to the headers directly, without the frustrations of path globbing or multiple include paths for the compiler. The directory names are mostly pretty self-explanatory:

AIModel, Aircraft, Airports, ATC, Autopilot, Cockpit, Controls (in the aircraft cockpit), Environment, FDM (only one constituent is in use), GUI (menus and the like), Include (for compiler configuration), Input (joysticks etc), Instrumentation, Main (initialization and command line parsing), Model (3D aircraft), MultiPlayer, Navaids, Network (data sharing), Objects (dynamically loaded and changing scenery), Replay, Scenery (static), Scripting (remote script control), Server, Sound, Systems and Time (in the simulated world).

FlightGear exposes the internal state of the simulation through the property database, part of the generic simulation infrastructure offered by *SimGear*[6]. This dynamically maps a name (such as `/position/latitude`) into an object with getter and setter methods. If the property will be accessed frequently, the pointer to the object can be stored so that the string lookup of the name only occurs one time. The single pointer indirection is still somewhat slower than hard linkage, but enhances the modularity of the code base. For example, the user interface definitions in XML files (panels, instruments, 3D animation, sound, etc) are able to refer to any property offered by any subsystem.

The simulator state is also accessible on the network. Adding the command line option `--telnet=5555` allows another computer (such as the instructor console) to interact with the simulator computer using a command such as `telnet simulator 5555`. This network interface allows any property in the database to be viewed or modified and includes remote enumeration of all the property names. This has been used to implement a complete external operator GUI and for automating FAA certification tests.

Many tasks within the simulator need to only run periodically. These are typically tasks that calculate values that don't change significantly in 1/60th of a second, but instead change noticeably on the order of seconds, minutes, or hours. Running these tasks every iteration would needless degrade per-

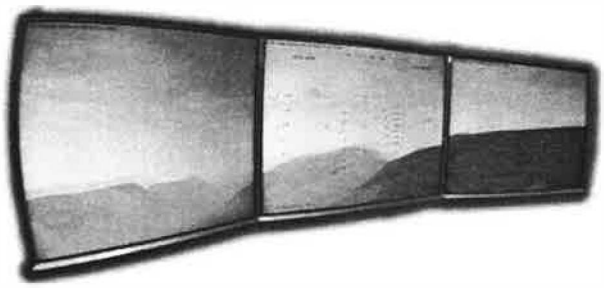


Figure 2: Panoramic scenery, configured by Curt Olson

formance. Instead, we would like to spread these out over time to minimize the impact they might have on frame rates, and minimize the chance of pauses and hesitations.

We do this using the *Event Manager and Scheduler*, which consists of two parts. The first part is simply a heap of registered events along with any management information associated with that event. The second part is a run queue. When events are triggered, they are placed in the run queue. The system executes only one pending event per iteration in order to balance the load. The manager also acquires statistics about event execution such as the total time spent running this event, the quickest run, the slowest run, and the total number of times run. We can output the list of events along with these statistics in order to determine if any of them are consuming an excessive amount of time, or if there is any chance that a particular event could run slow enough to be responsible for a perceived hesitation or pause in the flow of the simulation.

FlightGear itself supports threads for parallel execution, which distributes tasks such as scenery management over many iterations, but some of the library dependencies are not themselves thread clean. Thus all accesses to non-thread-clean libraries (such as loading a 3D model) need to be made by a single thread, so that other activities wishing to use the same library must be delayed. These small delays can lead to minor user-visible hesitations.

## Simulator Execution

FlightGear is packaged by all major distributions and most others too, so that installation of pre-built binaries can usually be completed in a few minutes. Almost all customization, including adding new aircraft and scenery, occurs through XML and through a structured directory tree. Most users can now simply use the prepackaged distributed binaries and no longer need to recompile the simulator to add new features.

However, this is a rapidly changing project and new functionality is added to the source code on an continuing basis. If a user wishes to take advantage of a new capability or function, when customizing the simulation for their individual needs, that source version does of course need to be compiled.

Installing and running FlightGear is relatively easy under Linux, especially compared to other operating systems with weak tool automation.

1. Install Linux normally and test Internet access.
2. Add video card support, using a maximum of 25% of memory for the 2D display, as 3D uses the remainder.
3. Enable hardware accelerated OpenGL support and test for speed, using *glTron*[5] for example.
4. Install[2] *PLIB* 1.8 or above, which is already in many distributions, and test with all the supplied examples to ensure all the API features are working.
5. Verify that headers for zlib and similar are present.
6. Download[6], compile and install *SimGear*.
7. While that compiles, download the *FlightGear* source.
8. With *SimGear* installed, compile and install *FlightGear*
9. While compiling, download FlightGear's base package. This contains data files that are required at runtime.
10. Type `runfgfs` and enjoy.

Starting from a blank hard drive with no operating system, FlightGear can be running in less than an hour.

## Simulating the Pilot's view

The new FlightGear pilot will probably not want to remain within the San Francisco bay area, which is the small scenery patch included in the Base package. The scenery server allows the selection and retrieval of any region of the world. Joining other users in the sky is another possibility.

Due to limited monitor size, the view that is available on a normal computer is a poor substitute for the wraparound windows of general aviation aircraft. This is especially true when the simulated aircraft has an open cockpit and an unrestricted view in almost all directions.

FlightGear can make use of multiple monitors to provide a nicer external view, possibly even wrap around, without special cabling. The additional computers and monitors need not be dedicated to this purpose. Once the command lines and fields of view (relative to the pilot) for each of the additional computers have been established, the main computer will make the necessary data available irrespective of whether those other computers are actually running FlightGear. In consequence, each of the additional computers can change from a 'cockpit window' to a office software workstation (for someone else) and, when available again, rejoin the FlightGear simulation session.

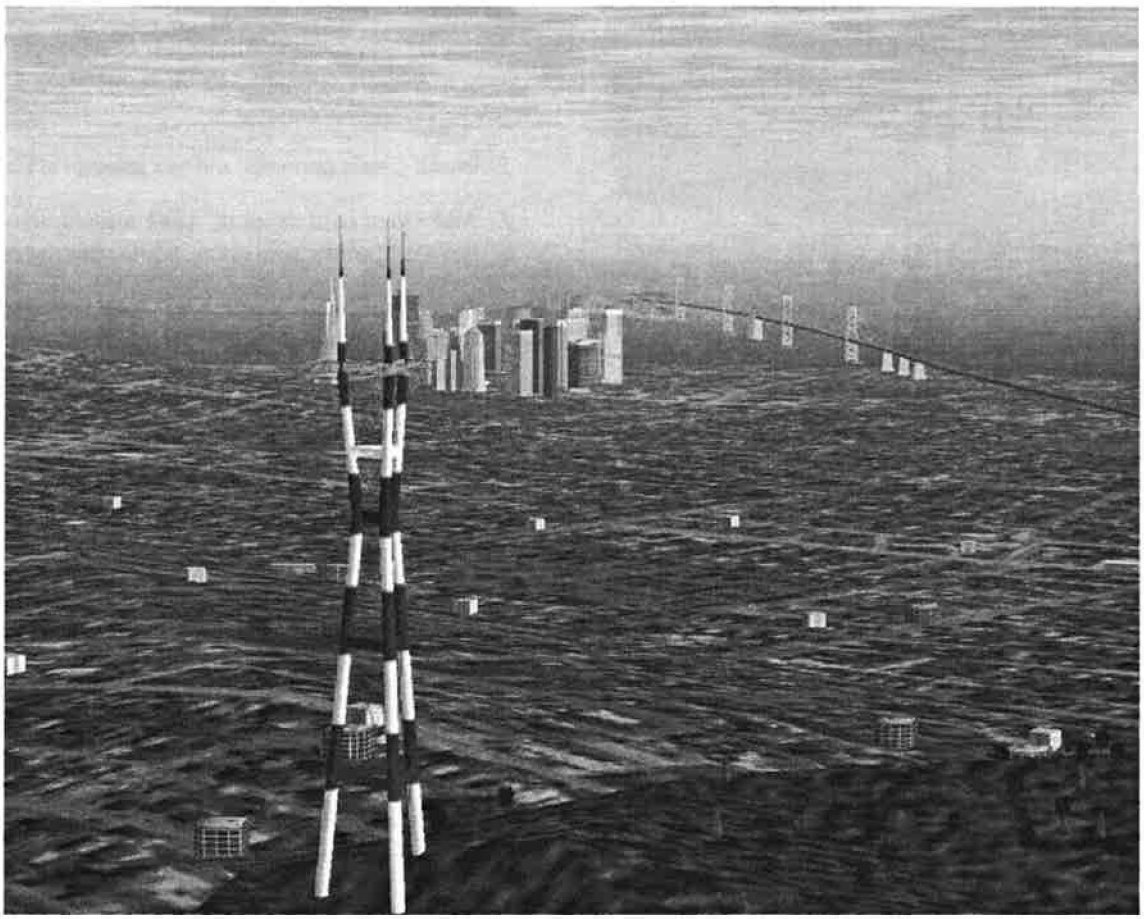


Figure 3: Looking east from Mount Sutro, just south of Golden Gate Park. The scenery shows the shoreline, variations in land use from scrubland on the hill and urban beyond, randomly placed trees in the scrubland and buildings in the urban area, custom created additional objects for the tower, downtown buildings and the Bay Bridge, and an interstate freeway. The sky shows the limited visibility due to haze and some scattered clouds.

FlightGear has built in support for network socket communication and the display synchronizing is built on top of this support. FlightGear also supports a null or do-nothing flight model which expects the flight model parameters to be updated somewhere else in the code. Combining these two features allows you to synchronize displays.

Here is how Curt Olson set up the example in figure 2:

1. Configure three near identical computers and monitors.
2. Pick one of the computers (i.e. the center channel) to be the master. The left and right will be slaves *s1* and *s2*.
3. When you start `runfgfs` on the master, use the command line options
   `--native=socket,out,60,s1,5500,udp`
   `--native=socket,out,60,s2,5500,udp`
   respectively to specify that we are sending the "native" protocol out of a udp socket channel at 60 Hz, to a slave machine on port 5500.
4. On each slave computer, the command line option `--native=socket,in,60,,5500,udp` shows that we expect to receive the native protocol via a udp socket on port 5500. The option `--fdm=external` tells the slave not to run it's own flight model math, but instead receive the values from an "external" source.
5. You need to ensure that the field of view on the scenery matches the apparent size of the monitor to the pilot. `--fov=xx.x` allows you to specify the field of view in degrees on each computer display individually.
6. `--view-offset=xx.x` allows you to specify the view offset direction in degrees. For instance,
   `--view-offset=0` for the center channel,
   `--view-offset=-50` for slave 1, and
   `--view-offset=50` for slave 2.

There is no built in limit to the number of slaves you may have. It wouldn't be too hard to implement a full $360^o$ wrap around display using 6 computers and 6 projectors, each covering $60^o$ field of view on a cylindrical projection screen. Ideally, the master computer should be chosen to be whichever visual channel has the lightest graphical workload. This might be the dedicated instrument panel, for example. If the master computer has a heavy graphical workload, the other channels will usually lag one frame behind. Select the graphics realism parameters to ensure that all the visual channels consistently achieve a solid and consistent frame rate (30 $Hz$ for example) and, if your video card supports it, lock the buffer swaps to the vertical refresh signal.

For optimal results, make sure the FOV each display subtends, matches the actual FOV that display covers from the pilot's perspective. From the top view, draw a line from the pilot's eye to each edge of the display. Measure the angle and use that for your visual channel configuration. Draw a line from the eye to the center of the display. Measure the angle from that line to the dead center straight ahead line. Use that angle for the view offset. This ensures that all objects in your simulator will be exactly life size.

## Simulating the Aircraft

The aerodynamic simulation may be only one constituent of the whole environment being simulated for the user, but its performance is critical to the quality of the user's simulation experience. Errors in this *Flight Dynamics Model* (FDM) are distracting to the pilot. Other simulator components, such as the autopilot, are designed to expect a realistic aircraft, may respond incorrectly as a result of FDM errors and provide additional pilot distractions. These factors can ruin the immersive experience that the user is seeking.

As a result of this concern, FlightGear abstracts all of the code that implements an FDM behind an object oriented interface. As future applications find that existing FDM choices do not meet their requirements, additional FDM code can be added to the project without impacting the consistent performance of existing applications.

The original FlightGear FDM was LaRCsim, originally modeling only a Navion, which currently models a Cessna 172 using dedicated C source that has the necessary coefficients hard coded. It is sufficient for most flight situations that a passenger would choose to experience in a real aircraft. Unusual maneuvers that are often intentionally performed for training purposes are poorly modeled, including deep stalls, incipient and developed spins and steep turns. The code also supports a Navion and a Cherokee, to a similar quality.

A research group at the University of Illinois created a derivative of LaRCsim, with simplified the models such that they are only really useful for cruise flight regimes. They enhanced the code with a parametric capability, such that a configuration file could be selected at simulation start to determine how the aircraft will fly. Their use for this modification was to investigate the effect on aircraft handling of progressive accumulations of ice.

Another group is developing a completely parametric FDM code base, where all the information is retrieved from XML format files. Their JSBSim project[7] can run independently of a full environmental simulation, to examine aerodynamic handling and other behavior. An abstraction layer links the object environment of FlightGear to the object collection of JSBSim to provide an integrated system. Among many others, this FDM supports the Cessna 172 and the X-15 (a experimental hypersonic rocket propelled research vehicle), providing the contrast between an aircraft used for teaching student pilots and an aircraft that could only be flown by trained test pilots.

An additional FDM code base, *YASim*, generates reasonable and flyable models for aircraft from very limited infor-

mation. This is especially valuable when a new aircraft is being added into FlightGear. Initially, there is often insufficient public data for a fully parametric model. This FDM allows the aircraft to be made available to the user community, thereby encouraging its users to find sources of additional data that will improve the model quality.

The rest of FlightGear's configuration files are now also XML, such as the engine models, the instrument panel layouts and instrument design, the HUD layout, the user preferences and saved state. The real benefit of using XML here is that people with no software development experience can easily and effectively contribute. Pilots, instructors, maintenance technicians and researchers each have in-depth technical knowledge of how a specific subsystem of an aircraft, and hence the simulator, should behave. It is critical that we allow them direct access to the internals that define their respective subsystem, without burdening them with having to dig through other subsystem data to get there. We have made huge progress on achieving this in the last two years.

## Simulating the Cockpit

In order to simulate the cockpit environment around the pilot, additional information is included from subsidiary XML files. These include a 3D model of the cockpit interior, associations of keyboard keys to panel switches, the instrument panel layout and position, visual representations of the individual instruments, mappings between joystick channels and flight controls, parametric descriptions of the head up display elements, a 3D model of the aircraft exterior, animation of moving aircraft surfaces and other reference data.

In the same way that aircraft manufacturers reuse much of their designs and instruments across product lines, many XML files are included by multiple aircraft models. Once loaded, this information is integrated into the simulation.

For example, one can look into the cockpit from outside, as shown in figure 4, and see the live instrument panel indications. This is the same exact instrument panel being shown to the pilot when inside the cockpit, demonstrating that FlightGear has a fully working 3D animated cockpit that is visible from inside or out.

Some aircraft types have 'glass cockpit' displays, as shown in figure 7. The independent project *OpenGC*[8] is a multiplatform, multi-simulator, open-source tool for developing and implementing high quality glass cockpit displays for simulated flightdecks. Due to the resolution and size limitations of computer monitors, the OpenGC images are best on a separate monitor from FlightGear.

The head up display of a real aircraft uses computer generated graphics, so the software can generally detect, and correct for, the flaws and inaccuracies in the sensors that are feeding it data. As a result, the information presented to the pilot is generally accurate. Simulating that is relatively easy,


Figure 4: Closeup of an A4 with the animated cockpit interior

since the actual state of the aircraft (in the properties) can be retrieved and directly displayed.

An important aspect of learning to fly an aircraft (without computer assistance) is understanding what the limitations and errors of the various instruments are, and when their indications can be trusted as useful flight data. Unfortunately, the information from panel instruments has errors, which in general only read a single sensor value with negligible correction for the limitations of the sensors being used. When the FlightGear panel advanced from no errors to having only two of the limitations implemented (VSI lag and compass turning errors), the non-pilot developers went from trivially flying instrument approaches to frequent ground impacts. Many more limitations have been realistically implemented since.

Considerable effort is needed to write this code. Gyroscopes can slow down and wobble, their rotation axis can drift, they can hit gimbal stops and tumble and their power source can be weak or fail. Air-based instruments are wrong in certain weather conditions, tend not to respond immediately, can be blocked by rainwater in the lines, or become unusable when iced over. Radio navigation is subject to line-of-sight, signals bounce off hills and bend near lake shores or where another aircraft is in the way and distant stations can interfere. Still more errors are associated with the magnetic compass, and other instruments that seem 'trivial'.

Currently, the communication radios are not implemented, so that pilots cannot use their microphone inputs to interact.

Radio usage is a large part of the complexity in operating at large and busy airports. Unfortunately, this often encourages pilots to fly the microphone and forget about the airplane, occasionally with disastrous results. We hope to implement this feature soon, to provide another source of challenging distractions to the pilot.

Although voice communication between pilots is not yet supported, there is an artificial intelligence (AI) subsystem that seeks to make the airspace feel less empty. This subsystem moves other aircraft around the sky as a source of distraction, issues ATC style instructions and responses, and ATIS messages. Somewhat surprisingly, everywhere in the world, the ATIS always has a British accent.

## Simulating the World

The purpose of the TerraGear project[9] is to develop open-source tools and rendering libraries and collect free data for building 3D representations (or maps) of the earth for use in real time rendering projects. There is much freely available *Geographic Information System* (GIS) data on the Internet. Because the core data for FlightGear has to be unrestricted, the default use of the project only uses source data that doesn't impose restrictions on derivative works. Three categories of data are used.

Digital Elevation Model (DEM) data is typically a set of elevation points on a regular grid. Currently, 30 arcsecond (about $1\,km = 0.6\,mi$) data for the whole world, and 3 arcsecond ($90\,m$) data from the Shuttle Radar Topography Mission (SRTM) for the United States and Eurasia, is available from the U.S. Geological Survey (USGS). Although the SRTM data was originally recorded in February 2000, the signal processing by the Jet Propulsion Laboratory (JPL) is being continuously improved. The recent releases with even finer grids, including 1 arcsecond, would offer much better resolution of landscape features but still suffer from artifacts around large buildings. Future improvements in these data sources are hoped for.

Irrespective of which data source is selected for a given area, an optimizing algorithm seeks to find the smallest number of flat triangles that provide a fairly smooth and realistic terrain contour. This algorithm reduces the number of triangles need to render an area while preserving all the detail within some specified error tolerance.

Other more specialized data such as airport beacon, lighthouse locations, radio transmission towers and the like are available in listings from various government agencies. These generally provide a short text description of the item and its geographic coordinates. The challenge is to convert each entry into a realistic visual object which can be inserted into the scenery database.

Polygonal data such as landmass outlines, lakes, islands, ponds, urban areas, glaciers, land use and vegetation are available from the USGS and other sources. Unfortunately, the land use data is many years old and thus may not be current with a pilot's local real world knowledge and this is not expected to change in the near future. The GSHHS database provides a highly detailed and accurate global land mass data so we can model precise coast lines for the entire world. Based on the source of the data and factoring in the land use data, we can select an appropriate texture which will be painted onto the individual triangles. Where necessary, triangles are subdivided to get the effect correct. Runways and taxiways are generated by converting the list of runway segments into polygons, painted with appropriate surface texture and markings, and then integrated into the scenery in the same way.

Clearly, someone can gain access to data sources that are under more restrictive licenses, use the TerraGear project tools to generate enhanced scenery and then distribute those files as they choose. Both the FlightGear and TerraGear projects encourage this kind of enhancement, because the basic open source packages cannot do this.

There is a trade-off between the quality of the scenery and the speed at which it can be rendered by the graphics card. As cards get faster, it becomes feasible to place more detail into the scenery while maintaining a useful and smooth visual effect. There are many techniques for adjusting the level of detail according to the altitude and attitude of the aircraft, to optimize the visual quality, but none of them are currently implemented as they cause visual artifacts.

The scenery system adds trees, buildings, lights (at night) and other objects. The choice of object, as well as the coverage density, is determined by the land use data. These objects are relatively slow to display in large quantities, so the user must trade off the reduction in display responsiveness against the improved cues for height, speed and underlying terrain. Like other settings, this property may be adjusted in real time. Figure 3 shows randomly placed buildings and trees, with a maximum range of about half way to downtown San Francisco, together with the manually placed downtown area and bay bridge.

Airports have runways and taxiways that are automatically created from the best available information to ensure that their locations, dimensions and markings correspond to real life. This is not trivial, since there are dozens of different levels of painting complexity in use. This information is also used to determine the pattern of lighting, if any - since some airports have no lights, that is shown at night and during twilight. Some lights are directional, multicolored, flashing or are defined to have a specific relative brightness. Figure 1 shows the lights and markings for KSFO.

Currently, the visual effect is clearly synthetic, as can be seen in figures 3 and 4, but it has sufficient information to readily permit navigation by pilotage (i.e. comparing the view out of the window to a chart). The compressed data requires about one kilobyte per square kilometer. All the information inside the scenery database is arranged in a four-

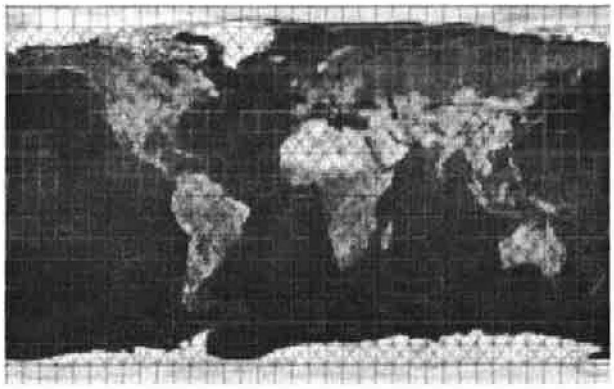


Figure 5: World scenery

level hierarchy, each level changing scale by a factor between 10 and 100:

1. One planet, currently only the Earth
2. $10^o \times 10^o$ rectangle as shown in figure 5,
3. $1^o \times 1^o \approx 70\,mi \times 50\,mi = 100\,km \times 60\,km$,
4. $50\,mi^2 = 100\,km^2$ approximately.

One of the difficulties facing the TerraGear developers is that most information sources are only generated at a national level. It is easy to justify writing special code to read and process data files for the largest ten countries, since they cover most of the land surface of the planet, but this approach rapidly reaches the point of diminishing returns.

There are already many organizations that painstakingly collect and transform the data into standardized formats, precisely for these kinds of applications. However, the huge amount of effort involved requires them to keep the prices extremely high in order to fund the conversions. Therefore, in the medium term, it is possible that these organizations (or one of their licensees) may start selling TerraGear compatible scenery files that is derived from their data archive. You can expect a high price tag for such reliable data though.

## Downloading the World

The scenery for the entire world currently requires 3 DVD-ROMs, which is a significant download for users with broadband access and an prohibitive barrier for dial up access users.

It was hoped that someone would get around to writing a utility for on-demand streaming of scenery to the user, but this hasn't happened. A significant factor is that this streaming real time bandwidth is much more expensive to host than the existing bulk retrieval. Money aside, it isn't a difficult problem.

Suppose we consider the pilot's viewpoint. Most general aviation aircraft cruise below 200 knots and flight visibility is (in real life) usually below 20 miles at their cruise altitudes. The database uses about one megabyte for 600 square miles so the peak streaming rate would be 12 megabytes/hour, less for areas previously visited. A 56K modem is easily capable of 12 megabytes/hour.

The utility for streaming scenery download does not need to be integrated into the core FlightGear source code. The latitude and longitude of the aircraft are already exported for use by independent programs, so the center of interest is trivially available. Since the scenery is stored in 100 $km^2$ pieces, an independent program need only generate a list of the closest elements that have not been fetched yet, and issue a `wget` to ensure that they will be available before the aircraft gets close enough for the pilot to see them.

## Simulating the Charts

Laptop/PDA applications for use in flight are becoming increasingly popular with light aircraft pilots, since they assist in situational awareness and in managing flight plan logs, navigation data and route planning. While *FlightMaster*, *CoPilot* and other applications are valuable tools, it is dangerous for pilots to use them in an aircraft without first becoming familiar with the user interface and gaining some practice.

FlightGear offers several specialist interfaces, one of which emits a stream of NMEA compliant position reports (the format used by GPS units) to serial port or UDP socket. This can be fed directly into one of those applications, which doesn't notice that this isn't coming from a real GPS, enabling the user to practice realistic tasks in the context of the simulated aircraft and all the realistic workload of piloting.

Data that is released into the public domain is generally of reduced quality, or out of date, or does not give widespread area coverage. The *TerraGear* scenery from such data is actually wrong, compared to the real world, but generally only in ways that are visually unobtrusive to the casual user.

These errors are much more visible in electronic navigation, such as needed for instrument flight, since the route tolerances are extremely tight. Navigating the simulated aircraft around imperfect scenery according to current Jeppesen (or NOS, etc) charts (or electronic databases) can be extremely frustrating and occasionally impossible when a piece of scenery is in the way.

To avoid the frustration, the *Atlas* project[10] has developed software which automatically synthesizes aviation style charts from the actual scenery files and databases being used by FlightGear. These charts, while inaccurate to the real world and therefore useless for flight in an aircraft, are extremely accurate for the simulated world in which the FlightGear aircraft operate. Thus, it is often easier to make printouts from the `Map` program of the *Atlas* project.

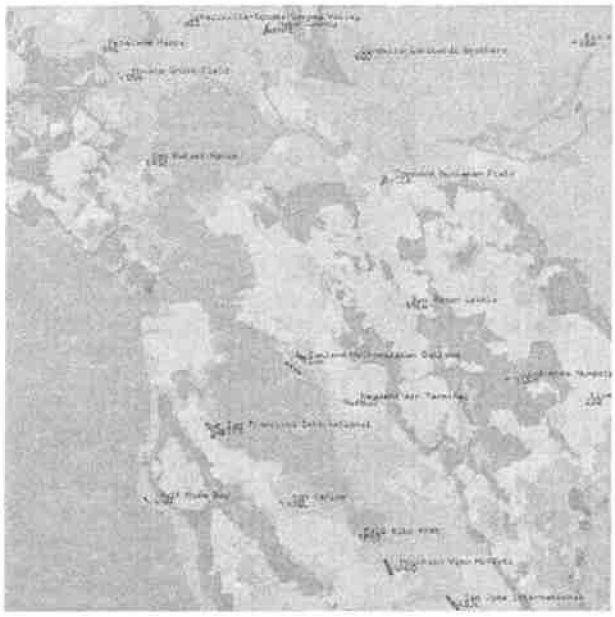


Figure 6: *Atlas* chart of San Francisco, California

The project also includes the namesake *Atlas* application. This can be used for browsing those maps and can also accept the GPS position reporting from FlightGear in order to display aircraft current location on a moving map display. This capability must be used selectively by the simulator pilot, since most small aircraft do not have built in map displays.

The *Atlas* moving map need not run on the same computer as the simulator, of course. It is especially valuable running on the instructor's console, where the pilot cannot see the picture, for gauging the student performance at assigned tasks.

## Simulating the Weather

Weather consists of many factors. Some items, such as air temperature and pressure, are invisible but have a strong effect on aircraft performance. Other items, such as smog layering, have no effect on the aircraft or piloting duties but contribute to realism (in Los Angeles, for example). In between these limits are many other items, such as wind and cloud, that must be simulated in order to reproduce the challenges facing the aircraft and pilot. Some complex items, such as turbulence, affect the simulation in many ways and are capable of making the aircraft realistically unflyable.

While FlightGear supports all these items, each of which can vary by location, altitude and time, leading to the difficulty of enabling the user to explain the desired configuration without too much effort. Three modes are currently available.

First, a single set of conditions can be specified on the command line which will be applied to the entire planet and do not change over time. This is very convenient for short duration and task specific uses, such as flying a single instrument approach from the IAF to the airport, where the same task will recur for each successive student session.

Second, all the weather configuration is accessible through the property database and so can be tweaked by the instructor (for example). This is useful for training on weather decision making, such as choosing between VFR, SVFR, IFR during deteriorating conditions.

Third, a background thread can monitor current weather conditions from `http://weather.noaa.gov` for the closest station. This is useful when conditions may be too dangerous to fly into intentionally, yet the pilot seeks experience with them. Such training, often an opportunity when a training flight is canceled, addresses the situation where the pilot had taken off before the weather deteriorated. Unfortunately, the transitions in weather conditions are necessarily harsh because the official weather reports may be issued as infrequently as once per hour. In any case, when flying between airports, the thread must at some point switch from old airport's report to the one ahead.

None of those is the 'correct' approach. All of them are especially suitable for specific situations. Other weather management approaches can be quickly created, if needed, since all the weather configuration parameters are properties and thus can be managed and modified across the network from a small specially-created utility.

The FlightGear environmental lighting model seeks to offer the best image that can be achieved with the limited dynamic range of computer monitors. For dusk and night flights, as shown in the left side of figure 1, it is best to use a darkened room in order that the subtle differences between the dark grays and blacks can be seen.

## Applications for the Simulator

We have a wide range of people interested and participating in this project. This is truly a global effort with contributors from just about every continent. Interests range from building a realistic home simulator out old airplane parts, to university research and instructional use, to simply having a viable alternative to commercial PC simulators.

## The Aberystwyth Lighter Than Air Intelligent Robot (ALTAIR)

The Intelligent Robotics Group at the University of Wales, Aberystwyth, UK is using FlightGear as part of their aerobot research[11] to design aerial vehicles that can operate in the atmosphere of other planets.

For those planets and moons that support an atmosphere (e.g. Mars, Venus, Titan and Jupiter), flying robots, or aerobots, are likely to provide a practical solution to the problem of extended planetary surface coverage for terrain mapping and surface/subsurface composition surveying. Not only

could such devices be used for suborbital mapping of terrain regions, but they could be used to transport and deploy science packages or even microrovers at different geographically separate landing sites.

The technological challenges posed by planetary aerobots are significant. To investigate this problem the group is building a virtual environment to simulate autonomous aerobot flight.

The NaSt3DGP computational fluid dynamics (CFD) software package generates meteorological conditions, which are 'loaded' into the FlightGear simulator to create realistic wind effects acting upon an aerobot when flying over a given terrain. The terrain model used by both FlightGear and NaSt3DGP is obtained from the MGS Mars Orbiter Laser Altimeter (MOLA) instrument, and the Mars Climate Database (MCD) is used to initialize the CFD simulation.

## University of Tennessee at Chattanooga

UTC has been using Flightgear as the basis of a research project started in August, 2001, with the goal of providing the Challenger Center at the university (and hopefully other centers in the future) a low cost virtual reality computer simulation.

The project is using flightgear and JSBSim, specifically the shuttle module, to develop a shuttle landing simulator. They are aiming to contribute instructions, on how to interface their virtual reality hardware with Flightgear, back to the OS community. The project is funded by the Wolf Aviation Foundation[12]. Dr. Andy Novobiliski is heading the research project.

## ARINC

Todd Moyer of ARINC used FlightGear as part of an effort to test and evaluate Flight Management Computer avionics and the corresponding ground systems. Certain capabilities of the Flight Management Computer are only available when airborne, which is determined by the FMC according to data it receives from GPS and INS sensors.

They wrote additional software that translates the NMEA output of FlightGear (including latitude, longitude, and altitude) into the ARINC 429 data words used by GPS and INS sensors. These data words are fed to the Flight Management Computer. the position information from FlightGear is realistic enough to convince the FMC that it is actually airborne, and allows ARINC to test entire 'flights' with the avionics.

## MSimulation

Marcus Bauer and others worked on a simulator cockpit environment using FlightGear as the software engine to drive a real cockpit, including three cockpit computers.

## Space Island

Space Island[13] Space Island are using FlightGear as the software for a moving cockpit entertainment simulator that supports both flight and space environments.

## Other applications

Many applications started using FlightGear years ago:

1. University of Illinois at Urbana Champaign. FlightGear is providing a platform for icing research for the Smart Icing Systems Project[14].

2. Simon Fraser University, British Columbia Canada. Portions of FlightGear were used in simulation to develop the needed control algorithms for an autonomous aerial vehicle.

3. Iowa State University. A senior project intended to retrofit some older sim hardware with FlightGear based software.

4. University of Minnesota - Human Factors Research Lab. FlightGear brings new life to an old Agwagon single seat, single engine simulator.

5. Aeronautical Development Agency, Bangalore India. FlightGear is used as as the image generator for a flight simulation facility for piloted evaluation of ski-jump launch and arrested recovery of a fighter aircraft from an aircraft carrier.

6. Veridian Engineering Division, Buffalo, NY. FlightGear is used for the scenery and out-the-window view for the Genesis 3000 flight simulator.

## Configuration User Interfaces

Scripting languages such as Python and Perl use a wrapper API for the network interface, hiding the protocol and operating system. This approach enables instructor interfaces to be developed in accordance with regulatory requirements and quickly customized to meet specific local needs.

## Simulating Flight Training

FlightGear could also be helpful when learning to fly aircraft. Flight training is carefully regulated by the government, to ensure that aircraft generally stay in the sky until their pilot intends for them to come down safely. There are thus some real concerns which need to be addressed before authorities can approve a system.

1. Do the controls feel, and operate, sufficiently like the ones in the aircraft that a pilot can use them without confusion? Are they easier to use and/or do they obscure dangerous real-life effects?

2. Does the software provide a forward view that is representative for the desired training environment?

3. Are the instruments drawn such that a pilot can easily read and interpret them as usual? Do they have the systematic errors that often cause accidents?

4. Are the cockpit switches and knobs intuitive to operate?

5. Operating within the limited envelope of flight configurations that is applied to the training activity, does it match the manufacturer's data for aircraft performance?

6. Are weather settings accessible to the instructor and sufficiently intuitive that they can change them quickly?

7. Are there easy mechanisms for causing the accurate simulation of system failures and broken instruments?

8. Can the pilot conduct normal interactions with air traffic control? Can the instructor easily determine whether the pilot is complying with the control instructions and record errors for subsequent review?

9. Is the pilot's manual for the simulator similar in content and arrangement to that of the aircraft being represented, such that it can readily be used in flight by the pilot?

10. Can all maneuvers be performed in the same way?

In that (partial) list of concerns, the quality of the actual flight simulation (which is really what FlightGear is offering) is a minor topic and and acceptable performance is easily achieved. In contrast, a large package of documentation must be added to the software to explain and teach people how to use it correctly. This has led to a number of separate projects whose goals are to meet or exceed the standards created by the United States Federal Aviation Administration (FAA).

It is easy to suggest that the FAA is being unrealistic in requiring this documentation, but they are responding to important traits in human nature that won't go away just because they're inconvenient.

For example, the things learnt first leave an almost unshakeable impression and, at times of severe stress, will over-rule later training. Thus, any false impressions that are learned by a beginning student through using a simulator will tend to remain hidden until a dangerous and potentially lethal situation is encountered, at which time the pilot may react wrongly and die. Pilots who use a simulator on an ongoing basis to hone their skills will get an excessively optimistic opinion of their skills, if the simulator is too easy to fly or

Figure 7: Example display from the OpenGC[8] project

does not exhibit common flaws. As a result, they will willingly fly into situations that are in practice beyond their skill proficiency and be at risk.

Clearly, a flight simulator (such as FlightGear) can only safely be used for training when under the supervision of a qualified instructor, who can judge whether the learning experience is beneficial. The documentation materials are essential to supporting that role.

## What's in the future?

In many areas of the project, the source code is stable and any ongoing programming rarely affects the interfaces used by XML files. The majority of the current developer effort centers around the crafting of nice looking 3D aircraft, animating their control surfaces, synthesizing appropriate sound effects, implementing an interactive cockpit, and adding detail to the aerodynamics parameters. The aerodynamic models are not (yet) accurate enough for use in all flight situations, so they don't reflect the challenges and excitement of acrobatic maneuvering.

Surround projectors, head mounted displays, directional sound and cockpit motion are rapidly converging into consumer technologies. Maybe we can immerse the users so well that they fly conservatively because they forget that they're not in real danger.

Aircraft wake is invisible, can last five minutes, descends slowly or spreads across the ground, is blown around by the wind and is extremely dangerous to following aircraft. A future extension to `fgd` could keep track of the hundreds of miles of wake trails in a given area and notify individual aircraft when they are encountering invisible severe turbulence.

Replication and scalability is only starting to take hold in the desktop environment. A room of several hundred computers acting as X terminals for word processing can reboot and, within a couple of minutes, all be running FlightGear identically. They're ready for the next class of student pilots.

## Conclusions

On the surface, FlightGear is a simple Open Source project that builds on many existing projects in the community traditions. Due to the subject it addresses, many issues and concerns are raised that rarely inconvenience most other project teams. These elements are providing the exciting challenges and variety of associated activities that the developer team is enjoying.

## About the Author

*Alexander Perry* holds M.A. and Ph.D. degrees in engineering from Cambridge University in England. A member of the IEEE Consultants Network in San Diego, he is one of the *FlightGear* developers, a commercial and instrument rated pilot, ground instructor and an aviation safety counselor in San Diego and Imperial counties of California.

## References

[1] http://www.flightgear.org/

[2] http://plib.sourceforge.net/

[3] http://www.linuxbase.org/

[4] http://oss.sgi.com/projects/ogl-sample/ABI/

[5] http://www.gltron.org/

[6] http://www.simgear.org/

[7] http://jsbsim.sourceforge.net/

[8] http://opengc.sourceforge.net/

[9] http://www.terragear.org/

[10] http://atlas.sourceforge.net/

[11] http://users.aber.ac.uk/dpb/aerobots.html

[12] http://www.wolf-aviation.org/

[13] http://www.spaceisland.de/

[14] http://www.aae.uiuc.edu/sis/mainpapers.html

[15] http://fgatd.sourceforge.net/

# Making RCU Safe for Deep Sub-Millisecond Response Realtime Applications

Dipankar Sarma
*Linux Technology Center*
*IBM Software Lab, India*
dipankar@in.ibm.com

Paul E. McKenney
*Linux Technology Center and Storage Software Architecture*
*IBM Corporation*
*Beaverton, OR 97006*
Paul.McKenney@us.ibm.com
http://www.rdrop.com/users/paulmck

## Abstract

Linux™ has long been used for soft realtime applications. More recent work is preparing Linux for more aggressive realtime use, with scheduling latencies in the small number of hundreds of microseconds (that is right, *micro*seconds, *not* milliseconds). The current Linux 2.6 RCU implementation both helps and hurts. It helps by removing locks, thus reducing latency in general, but hurts by causing large numbers of RCU callbacks to be invoked all at once at the end of the grace period. This batching of callback invocation improves throughput, but unacceptably degrades realtime response for the more discerning realtime applications.

This paper describes modifications to RCU that greatly reduce its effect on scheduling latency, without significantly degrading performance for non-realtime Linux servers. Although these modifications appear to prevent RCU from interfering with realtime scheduling, other Linux kernel components are still problematic. We are therefore working on tools to help identify the remaining problematic components and to definitively determine whether RCU is still an issue. In any case, to the best of our knowledge, this is the first time that anything resembling RCU has been modified to accommodate the needs of realtime applications.

## 1 Introduction

Tests of realtime response on the Linux 2.6 kernel found unacceptable scheduling latency, in part due to the batching of callbacks used in the RCU implementation. This batching is essential to good performance on non-realtime servers, since the larger the batch, the more callbacks the overhead of detecting an RCU grace period may be amortized over. However, because these callbacks run in a tasklet that runs at softirq level, callback processing cannot be preempted. Since heavy loads can result in well over a thousand RCU callbacks per grace period, RCU's contribution to scheduling latency can approach 500 microseconds, which far exceeds the amount that can be tolerated by some classes of realtime applications. Furthermore, extreme denial-of-service workloads have been observed to generate more than 30,000 RCU callbacks in a single grace period, which would result in a correspondingly greater degradation of scheduling latency. This situation motivated some modifications to RCU, with the goal of eliminating RCU's contribution to the excessive scheduling latency.

This paper presents some background on RCU in Section 2, describes the problem that was causing excessive scheduling latency in Section 3, discusses three proposed solutions in Section 4, and evaluates the three solutions in Section 5.

## 2 RCU Background

RCU is a reader-writer synchronization mechanism that takes asymmetric distribution of synchronization overhead to its logical extreme: read-side critical sections incur zero synchronization overhead, containing no locks, no atomic instructions, and, on most architectures, no memory-barrier instructions. RCU therefore achieves near-ideal performance for read-only workloads on most architectures. Write-side critical sections must therefore incur substantial synchronization overhead, deferring destruction and maintaining multiple versions of data structures in order to accommodate the read-side critical sections. In addition, writers must use some synchronization mechanism, such as locking, to provide for orderly updates. Readers must somehow inform writers when they finish so that writers can determine when it is safe to complete destructive operations.

In the Linux 2.6 kernel, RCU signals writers by non-atomically incrementing a local counter in the context-switch code. If this is a given CPU's first such increment for the current grace period, then the CPU clears its bit from a global bitmask. If it is the last CPU to clear its bit, then the end of the grace period has been reached, and RCU callbacks may safely be invoked.

The actual implementation is more heavily optimized than is described here. More details are available else-

where [ACMS03, MSA+02, McK03]. The performance benefits of RCU in the Linux kernel are also well documented [MAK+01, LSS02, MSS04, McK04], and benefits of RCU and of similar synchronization techniques in other environments have been published as well [KL80, ML84, Pug90, MS98, GKAS99, Sei03, SAH+03].

## 3 RCU Scheduling Latency Problem

The `amlat` test program runs a realtime task that schedules itself to run at a specific time. The `amlat` test program then measures how much the actual time is delayed from that specified.

In one test of a small configuration under heavy load, 1,660 callbacks were queued to be executed at the end of a single grace period, resulting in a scheduling latency of 711 microseconds on a single-CPU 2.4GHz x86 system. This far exceeds the goal of 250 microseconds.

The heavy load consisted of filesystem and networking operations, which resulted in large numbers of RCU callbacks being scheduled from the dcache and IP route cache subsystems.

Note that RCU callbacks are executed in the context of a tasklet, which runs either in interrupt context or in the context of the "ksoftirqd" kernel-daemon process. However, `do_softirq()`, which actually invokes the `rcu_process_callbacks()` function, uses a combination of `local_irq_save()` and `local_bh_disable()`, which has the effect of disabling preemption across the invocation of all RCU callbacks, even when running in ksoftirqd context.

Large numbers of RCU callbacks can therefore degrade realtime scheduling latency, as shown in Figure 1. In this figure, two CPUs go through a grace period while scheduling RCU callbacks. Each CPU's set of RCU callbacks is executed from `rcu_do_batch()` in softirq context after the end of the grace period, which directly increases the realtime scheduling latency, as shown in the lower right portion of the figure. This situation raises the question of what might be done to mitigate this latency increase, thereby preventing degradation of realtime response.

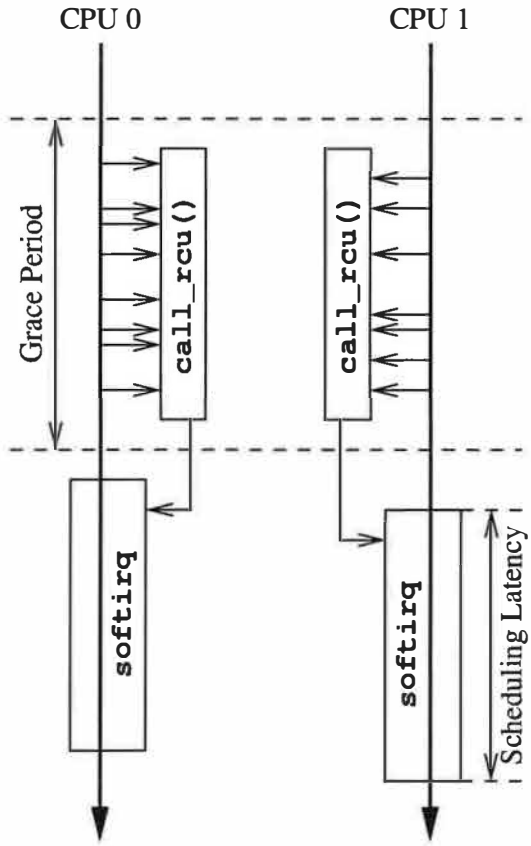



Figure 1: RCU Degrading Realtime Scheduling Latency

## 4 RCU Scheduling Latency Solutions

One could also imagine solving this problem by going back to traditional locking primitives, but this would impose unacceptable performance degradation and scaling limitations on Linux servers. We therefore resolved to solve the scheduling-latency problem in such a way that RCU could be used in realtime environments.

Thus far, we are investigating three solutions to this problem:

1. Providing per-CPU kernel daemons to process RCU callbacks when there are too many to process

```
1 int
2 rq_has_rt_task(int cpu)
3 {
4   struct runqueue *rq = cpu_rq(cpu);
5   return (sched_find_first_bit(rq->active->bitmap) <
6           MAX_RT_PRIO);
7 }
```

Figure 2: Functions Encapsulating Per-CPU Realtime-Task Count

at softirq level.

2. Directly invoking the RCU callback in those cases where it is safe to do so, rather than queuing the callback to be executed at the end of the next grace period.
3. Throttling RCU callback invocation so that only a limited number are invoked at a given time.

The first and last of these solutions require a mechanism that determines when there is a runnable realtime task on a given CPU. Such realtime tasks may be detected by checking a given runqueue's `active` bitmap, as was suggested by Nick Piggin, and as shown in Figure 2. The three solutions are described at length in the next sections.

## 4.1 Per-CPU Kernel RCU-Callback Daemons

The per-CPU kernel RCU-callback daemons [Sar04a], or *krcud* for short, were inspired by the "rcu" implementation of the RCU infrastructure in the Linux 2.6 kernel [MSA+02]. The idea is to modify `rcu_do_batch()` to limit the number of callbacks processed at a time to the value in module parameter `rcupdate.bhlimit`, which defaults to 256, but only under the following conditions:

1. the kernel has been built with the `CONFIG_LOW_LATENCY` kernel parameter,
2. there is a runnable realtime task on this CPU, and
3. `rcu_do_batch()` is running from softirq context.

If either of the first two conditions do not hold, then there is no reason to limit latency on this CPU. If the last condition does not hold, then preemption will limit execution time as needed, so no explicit limit checking is required.

When limiting does occur in `rcu_do_batch()`, any excess callbacks are queued for processing by the CPU's *krcud* on that CPU's `rcudlist` CPU-local variable. These callbacks are added to the *head* of this list in order to avoid any possibility of callback starvation. Note that callbacks can be processed out of order when limiting is in effect, since `rcu_do_batch()` can be invoked from the softirq context at the end of a grace period, even when *krcud* is running. We do not know any situation where such reordering is harmful, but strict ordering can be easily enforced should such a situation arise.

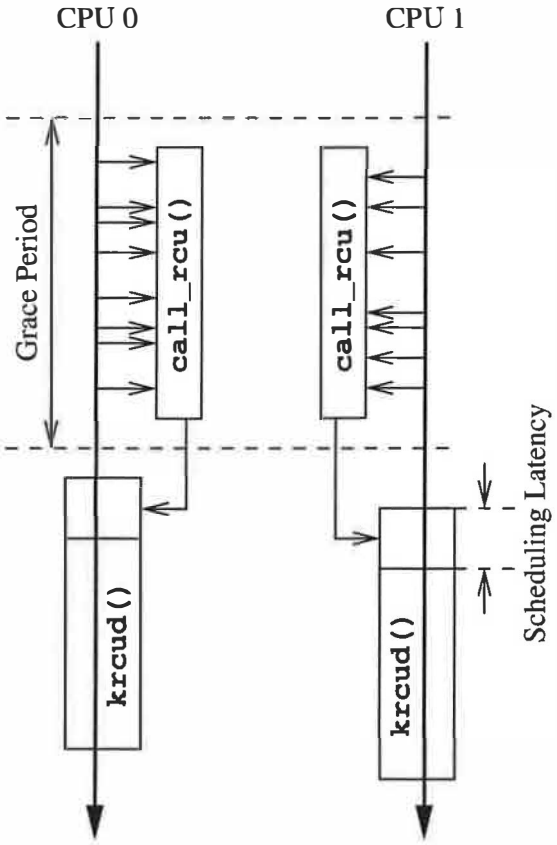



Figure 3: krcud Preserves Realtime Scheduling Latency

Since *krcud* is fully preemptible, the situation is as shown in Figure 3. The first few RCU callbacks are invoked from softirq context, which cannot be preempted. The execution time of these few RCU callbacks thus degrade realtime scheduling latency, but only slightly, as any additional RCU callbacks are invoked from *krcud* context, which is fully preemptible.

The `rcu_do_batch()` function, which invokes RCU callback, but limits the callback batch size when run from softirq context, is shown in Figure 4. Line 6 captures the current CPU. Note that (for once) `get_cpu()` is not needed:

1. If invoked from `krcud()`, execution is forced to remain on a single CPU via a `set_cpus_allowed()` call.
2. If invoked from `rcu_process_callbacks()`, preemption is disabled due to running in softirq context.

Line 7 invokes `rcu_bh_callback_limit()` in order to determine the maximum number of callbacks that may be executed, which is `bhlimit` if we are running in softirq context and there is a runnable realtime

```
 1 static void rcu_do_batch(struct list_head *list)
 2 {
 3   struct list_head *entry;
 4   struct rcu_head *head;
 5   unsigned int count = 0;
 6   int cpu = smp_processor_id();
 7   unsigned int limit = rcu_bh_callback_limit(cpu);
 8
 9   while (!list_empty(list)) {
10     entry = list->next;
11     list_del(entry);
12     head = list_entry(entry, struct rcu_head, list);
13     head->func(head->arg);
14     if (++count > limit && rq_has_rt_task(cpu)) {
15       list_splice(list, &RCU_rcudlist(cpu));
16       wake_up_process(RCU_krcud(cpu));
17       break;
18     }
19   }
20 }
```

Figure 4: Limiting RCU Callback Batches

```
1 static inline unsigned int
2 rcu_bh_callback_limit(int cpu)
3 {
4   if (in_softirq() && RCU_krcud(cpu))
5     return bhlimit;
6   return (unsigned int)-1;
7 }
```

Figure 5: Determining Maximum RCU Callback Batch Size

task on this CPU, or a very large integer otherwise, as can be seen in Figure 5. Lines 9-19 look over the callbacks, invoking each in turn until either the list is empty or the maximum allowable number has been exceeded. Lines 10 and 11 remove the first element from the list, and line 12 obtains a pointer to the `struct rcu_head`. Line 13 then invokes the RCU callback. Lines 14-18 check for exceeding the limit, but only if there is a runnable realtime task on this CPU. If there is, line 15 prepends the remainder of the list to this CPU's list of callbacks that are waiting for `krcud()`, line 16 wakes up this CPU's `krcud()`, and line 17 exits the "while" loop.

A given CPU's krcud task is created when that CPU is first brought online by `rcu_cpu_notify`, as shown in Figure 6. CPUs are brought up in two stages, the `CPU_UP_PREPARE` stage and the `CPU_ONLINE` stage. The `CPU_UP_PREPARE` stage is handled by lines 7-9, which invoke `rcu_online_cpu()`, which in turn initializes the RCU per-CPU data and initializes the per-CPU tasklet that processes callbacks when limiting is not in effect. At boot time, `rcu_online_cpu()` is instead called from `rcu_init()`. The `CPU_ONLINE` stage is handled by lines 10-13, which invoke `start_krcud()`, which starts the krcud tasks if appropriate. At boot time, `start_krcud()` is instead called from `rcu_late_init()`, which is registered for execution via `__initcall()`. Failure to start the

```
 1 static int __devinit
 2 rcu_cpu_notify(struct notifier_block *self,
 3                unsigned long action, void *hcpu)
 4 {
 5   long cpu = (long)hcpu;
 6   switch (action) {
 7   case CPU_UP_PREPARE:
 8     rcu_online_cpu(cpu);
 9     break;
10   case CPU_ONLINE:
11     if (start_krcud(cpu) != 0)
12       return NOTIFY_BAD;
13     break;
14   /* Space reserved for CPU_OFFLINE :) */
15   default:
16     break;
17   }
18   return NOTIFY_OK;
19 }
```

Figure 6: Creating krcud Tasks: `rcu_cpu_notify`

```
 1 static int start_krcud(int cpu)
 2 {
 3   if (bhlimit) {
 4     if (kernel_thread(krcud, (void *)(long)cpu,
 5                       CLONE_KERNEL) < 0) {
 6       printk("krcud for %i failed\n", cpu);
 7       return -1;
 8     }
 9
10     while (!RCU_krcud(cpu))
11       yield();
12   }
13   return 0;
14 }
```

Figure 7: Creating krcud Tasks

krcud task results in failure to start the CPU.

The `start_krcud()` function starts a krcud task for a specified CPU, and is shown in Figure 7. If the module parameter `bhlimit` is non-zero, the kernel thread is created by lines 4-8. Lines 10-12 then wait until the newly created krcud has initialized itself and is ready to accept callbacks. This function returns 0 on success and -1 on failure.

The `krcud()` function processes callbacks whose execution has been deferred, and is shown in Figure 8. Unlike the tasklets used by the 2.6 RCU infrastructure, `krcud()` invokes the RCU callbacks preemptibly, so that RCU callback execution from `krcud()` cannot degrade realtime scheduling latency. Note that each `krcud()` runs only on its own CPU, so that RCU callbacks are guaranteed never to be switched from one CPU to another while executing.

Line 3 of `krcud()` casts the argument, and line 4 converts this task to a daemon, setting the name, discarding any user-mode address space, blocking signals, closing open files, and setting the init task to be the newly created task's parent. Line 5 sets the krcud task's priority to the highest non-realtime priority. Line 6 marks the krcud task as required for swap operations, and line 8 restricts the task to run only on the specified CPU. Line 10

```
 1 static int krcud(void * __bind_cpu)
 2 {
 3   int cpu = (int) (long) __bind_cpu;
 4   daemonize("krcud/%d", cpu);
 5   set_user_nice(current, -19);
 6   current->flags |= PF_IOTHREAD;
 7   /* Migrate to the right CPU */
 8   set_cpus_allowed(current, cpumask_of_cpu(cpu));
 9   BUG_ON(smp_processor_id() != cpu);
10   __set_current_state(TASK_INTERRUPTIBLE);
11   mb();
12   RCU_krcud(cpu) = current;
13   for (;;) {
14     LIST_HEAD(list);
15     if (list_empty(&RCU_rcudlist(cpu)))
16       schedule();
17     __set_current_state(TASK_RUNNING);
18     local_bh_disable();
19     while (!list_empty(&RCU_rcudlist(cpu))) {
20       list_splice(&RCU_rcudlist(cpu), &list);
21       INIT_LIST_HEAD(&RCU_rcudlist(cpu));
22       local_bh_enable();
23       rcu_do_batch(&list);
24       cond_resched();
25       local_bh_disable();
26     }
27     local_bh_enable();
28     __set_current_state(TASK_INTERRUPTIBLE);
29   }
30 }
```

Figure 8: krcud Function

marks the task as alive, line 11 executes a memory barrier to prevent misordering, and line 12 sets the CPU's `krcud` per-CPU variable to reference this krcud task. Lines 13-29 loop processing any RCU callbacks placed on the `rcudlist`. Lines 15-16 wait for RCU callbacks to appear on this list, and line 17 sets the task state to running. Line 18 masks interrupts (which are restored by line 27), and lines 19-26 loop processing the callbacks on this CPU's `rcudlist`. Lines 20-21 move the contents of this CPU's `rcudlist` onto the local `list` variable, at which point it is safe for line 22 to re-enable interrupts. Line 23 invokes `rcu_do_batch()` to invoke the callbacks, and, since we are calling it from krcud context, it will unconditionally invoke all of them, relying on preemption to prevent undue delay of realtime tasks. Line 24 yields the CPU, but only if there is some other more deserving task, as would be the case after timeslice expiration. Line 25 then disables interrupts, setting up for the next pass through the "while" loop. As noted earlier, line 27 re-enables interrupts. Line 28 sets up to block on the next pass through the "for" loop.

This approach limits the number of callbacks that may be executed by `rcu_do_batch()` from softirq context. The duration of a grace period protects against too-frequent invocations of `rcu_do_batch()`, which could otherwise result in an aggregate degradation of realtime response. Since `krcud()` runs with preemption enabled, it cannot cause excessive realtime response degradation, and, in addition, can handle any RCU callback load up to the full capacity of the CPU.

```
 1 #if !defined(CONFIG_SMP) && defined(CONFIG_LOW_LATENCY
 2 static inline void
 3 call_rcu_rt(struct rcu_head *head,
 4            void (*func)(void *arg), void *arg)
 5 {
 6   func(arg);
 7 }
 8 #else
 9 static inline void
10 call_rcu_rt(struct rcu_head *head,
11            void (*func)(void *arg), void *arg)
12 {
13   call_rcu(head, func, arg);
14 }
15 #endif
```

Figure 9: Uniprocessor Call-Through RCU

Further refinements under consideration include:

1. Use elapsed time rather than numbers of callbacks to enforce the limiting in `rcu_do_batch()`.
2. Dynamically varying the number of callbacks to be executed based on workload or other measurement.

## 4.2 Direct Invocation of RCU Callbacks

Traditionally, most realtime and embedded systems have had but a single CPU. Single-CPU systems can in some cases short-circuit some of the RCU processing in some cases.

For example, if an element has just been removed from an RCU-protected data structure, and if there are no references to this element anywhere in the call stack, the element may safely be freed, since there is no other CPU that can be holding any additional references. However, it is not always possible to determine whether the call stack is free of references. For example, interrupt handlers can interrupt any function that runs without masking interrupts. Furthermore, many functions are invoked via function pointers or APIs that might be used anywhere in the kernel.

Therefore, direct invocation of RCU callbacks cannot be applied in all cases. Each use of RCU must be inspected to determine whether or not that particular use qualifies for direct invocation. However, it turns out that the important cases of dcache and of the IP route cache do qualify. When running on a uniprocessor, these two subsystems can simply immediately execute the RCU callback, so that there is no "pileup" of RCU callbacks at the end of the grace period.

Figure 9 shows how a `call_rcu_rt()` primitive may be defined, which immediately invokes the RCU callback in a realtime uniprocessor kernel, but invokes `call_rcu()` otherwise [Sar03]. The new `call_rcu_rt()` API prevents existing `call_rcu()` users from breaking, while allowing specific subsystems to use RCU in a more realtime-friendly manner.

Given this primitive, the trivial change to `d_free()` shown in Figure 10 renders the dcache subsystem

```
1 static void d_free(struct dentry *dentry)
2 {
3   if (dentry->d_op && dentry->d_op->d_release)
4     dentry->d_op->d_release(dentry);
5    call_rcu_rt(&dentry->d_rcu, d_callback, dentry);
6 }
```

Figure 10: dcache Call-Through RCU

```
1 #define rcu_read_lock_bh()      local_bh_disable()
2 #define rcu_read_unlock_bh()    local_bh_enable()
```

Figure 11: Disabling softirq Processing

realtime-friendly. The single `call_rcu()` in dcache has simply been replaced by `call_rcu_rt()`.

The changes required to the IP route cache are more complex, due to the fact that the route cache may be updated from interrupt context, but is accessed from process context. For an example of the problem that this poses, suppose that `__ip_route_output_key()` is interrupted while accessing the IP route cache in process context, and that the interrupt handler invokes softirq upon return. A softirq action might then delete the entry that `__ip_route_output_key()` is currently referencing. If the interrupt handler were to invoke `call_rcu_rt()`, then `__ip_route_output_key()` would fail upon return from interrupt.

This problem can be solved by having `__ip_route_output_key()` disable softirq (and bottom-half processing) during the traversal, similar to the manner in which preemption is already disabled. New `rcu_read_lock_bh()` and `rcu_read_unlock_bh()` primitives do just this, as shown in Figure 11. The IP route cache code (in functions `rt_cache_get_first()`, `rt_cache_get_next()`, `rt_cache_get_next()`, `rt_cache_seq_next()`, `__ip_route_output_key()`, and `ip_rt_dump()`) is then changed to use these new operations in place of `rcu_read_lock()` and `rcu_read_unlock()`.

Finally, as with dcache, the `rt_free()` and `rt_drop()` functions are changed to use `call_rcu_rt()` instead of `call_rcu()`, as shown in Figure 12.

These changes are quite straightforward, but of course this `call_rcu_rt()` approach works only on single-CPU systems. The increasing popularity of multithreaded CPUs makes this restriction less tenable on x86 CPUs, though it would still hold on some embedded CPUs. In addition, existing and planned uses of `call_rcu()` must be carefully vetted in order to ensure that direct invocation of the RCU callback is safe. At this writing, dcache and IP route cache are the two biggest realtime offenders, and they both are amenable to use of `call_rcu_rt()`, but it is easy to imagine less

```
1 static __inline__ void rt_free(struct rtable *rt)
2 {
3   call_rcu(&rt->u.dst.rcu_head,
4            (void (*)(void *))dst_free, &rt->u.dst);
5 }
6
7 static __inline__ void rt_drop(struct rtable *rt)
8 {
9   ip_rt_put(rt);
10  call_rcu(&rt->u.dst.rcu_head,
11           (void (*)(void *))dst_free, &rt->u.dst);
12 }
```

Figure 12: Freeing and Dropping IP Route Table Entries

fortunate circumstances.

As a result, a realtime-friendly `call_rcu()` implementation would be preferable.

## 4.3 Throttling of RCU Callbacks

Another solution to the realtime-degradation problem is to throttle softirq, so that only a limited number of RCU callbacks may execute during a given invocation of `do_softirq()` [Sar04b]. This approach was independently suggested by Andrea Arcangeli, and is illustrated in Figure 13, where the callbacks are executed in short bursts, limiting the realtime scheduling-latency degradation.

This solution is implemented using two additional per-CPU variables, `RCU_donelist`, which is a list of RCU callbacks awaiting invocation, and `RCU_plugticks`, which counts down the number of jiffies to block RCU callback invocation. `RCU_plugticks` is decremented each scheduling clock tick on each CPU in `scheduler_tick()`. There are also two module parameters, `rcumaxbatch`, which is the maximum number of callbacks that may be executed in a single softirq invocation, and `rcuplugticks`, which is the number of jiffies to wait after exceeding the `rcumaxbatch` limit before resuming RCU callback invocation. Note that `rcuplugticks` may be set to zero, in which RCU callbacks can be run continuously, which allows easy experimentation.

This callback limiting is enforced in `rcu_do_batch()`, which is shown in Figure 14. The differences from the stock 2.6 kernel implementation are quite small. Lines 5 and 6 add `count` and `cpu` variables that count the number of RCU callbacks invoked and track the current CPU, respectively. Line 13 checks for too many RCU callback invocations and line 14 sets the per-CPU `RCU_plugticks` variable in order to prevent RCU callback invocation on this CPU for the next `rcuplugticks` jiffies. Line 15 checks to see if there is to be no such delay, and, if so, line 16 reschedules the tasklet.

The `rcu_process_callbacks()` function has

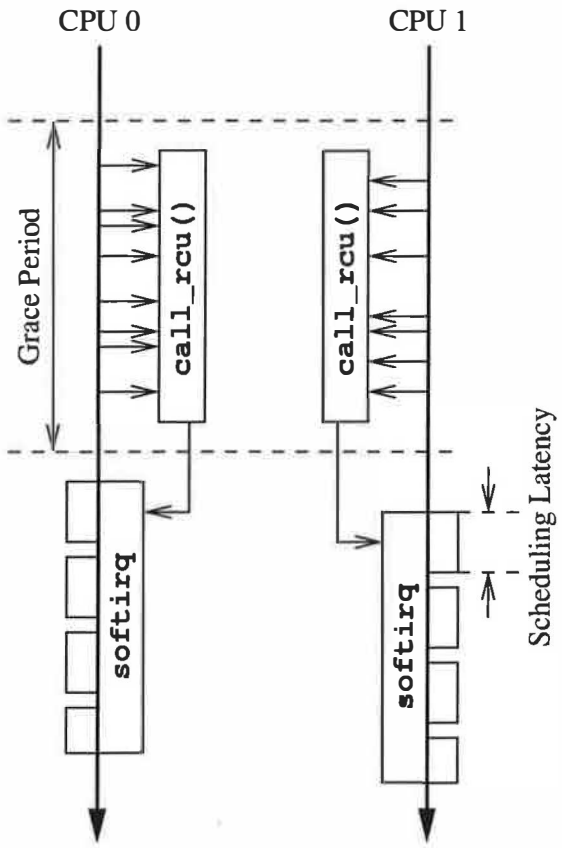



Figure 13: Throttling Preserves Realtime Scheduling Latency

```
 1 static void rcu_do_batch(struct list_head *list)
 2 {
 3   struct list_head *entry;
 4   struct rcu_head *head;
 5   int count = 0;
 6   int cpu = smp_processor_id();
 7
 8   while (!list_empty(list)) {
 9     entry = list->next;
10     list_del(entry);
11     head = list_entry(entry, struct rcu_head, list);
12     head->func(head->arg);
13     if (++count >= rcumaxbatch) {
14       RCU_plugticks(cpu) = rcuplugticks;
15       if (!RCU_plugticks(cpu))
16         tasklet_hi_schedule(&RCU_tasklet(cpu));
17       break;
18     }
19   }
20 }
```

Figure 14: Limiting RCU Callback Batch Size

```
@@ -153,18 +164,16 @@
   spin_unlock(&rcu_ctrlblk.mutex);
 }

-
 /*
  * This does the RCU processing work from
  * tasklet context.
  */
 static void
 rcu_process_callbacks(unsigned long unused)
 {
   int cpu = smp_processor_id();
-  LIST_HEAD(list);

   if (!list_empty(&RCU_curlist(cpu)) &&
       rcu_batch_after(rcu_ctrlblk.curbatch,
                       RCU_batch(cpu))) {
-    list_splice(&RCU_curlist(cpu), &list);
+    list_splice_tail(&RCU_curlist(cpu),
+                     &RCU_donelist(cpu));
     INIT_LIST_HEAD(&RCU_curlist(cpu));
   }

@@ -185,8 +194,8 @@
     local_irq_enable();
   }
   rcu_check_quiescent_state();
-  if (!list_empty(&list))
-    rcu_do_batch(&list);
+  if (!list_empty(&RCU_donelist(cpu)) &&
+      !RCU_plugticks(cpu))
+    rcu_do_batch(&RCU_donelist(cpu));
 }

 void rcu_check_callbacks(int cpu, int user)
```

Figure 15: Callback-Processing Changes

small modifications to place RCU callbacks that are ready to be invoked onto the per-CPU `RCU_donelist` list rather than on a local list, and to check for `RCU_plugticks`. The diffs are shown in Figure 15.

This small set of changes relies on the fact that `do_softirq()` exits after `MAX_SOFTIRQ_RESTART` number of iterations. When `do_softirq()` is invoked from `ksoftirqd()`, returning to `ksoftirqd()` re-enables preemption. On the other hand, when `do_softirq()` is invoked from interrupt context, returning to interrupt context in turn results in exiting interrupt context. Either alternative prevents `rcu_do_batch()` from excessively degrading realtime response.

## 5 Evaluation

These three approaches were tested on a uniprocessor 2.4GHz P4 system with 256MB of RAM running `dbench 32` in a loop. The kernel was built with `CONFIG_PREEMPT=y`, and the configuration excluded realtime-problematic modules such as VGA. Realtime scheduling latency was measured using Andrew Morton's `amlat` utility. The results are shown in Table 1. All three approaches greatly decrease realtime scheduling latency. Although direct invocation performs somewhat better than do the other two approaches, the differ-

ence is not statistically significant. Therefore, the simpler throttling approach seems preferable at present.

Although these numbers do not meet the 250-microsecond goal, they do indicate that RCU has been made safe for realtime environments. Changes to other parts of Linux will be needed in order to fully meet this goal. Such changes are likely to expose more significant performance differences between the three low-latency RCU approaches, so these tests should be re-run at that time.

Note that although the current testing techniques are not sufficient to validate the Linux 2.6 kernel for use by hard-realtime applications on which lives depend, they do demonstrate usefulness to soft realtime applications, even those requiring deep sub-millisecond realtime response.

## 6 Future Work

Future work includes applying realtime modifications to RCU in order to better withstand denial-of-service attacks, including taking full-network-adaptor-speed attacks while still providing good response to console input and user commands. It is likely that successfully withstanding such attacks will require additional work on the softirq layer in order to ensure that user processes are allowed to run even when the attack is sufficient to consume the entire system with softirq processing.

Of course, Linux will require more work if it is to meet more stringent realtime scheduling latencies, to say nothing of hard realtime requirements. Since some realtime applications require 10-microsecond scheduling latencies, it will be interesting to see if Linux can meet these applications' needs without sacrificing its usefulness to other workloads or its simplicity.

## 7 Acknowledgments

We owe thanks to Robert Love and Andrew Morton, who brought this problem to our attention. We are indebted to Andrew Morton for the `amlat` application that measures realtime scheduling latency, and to Jon Walpole and to Orran Krieger for many valuable discussions regarding RCU. We are grateful to Tom Hanrahan, Vijay Sukthankar, Daniel Frye, Jai Menon, and Juergen Deicke for their support of this effort.

## 8 Availability

RCU is freely available as part of the Linux 2.6 kernel from `ftp://kernel.org/pub/linux/kernel/v2.6`. The patches described in this paper are freely available from any archive of the Linux Kernel Mailing List.

## Legal Statement



## References

[ACMS03] Andrea Arcangeli, Mingming Cao, Paul E. McKenney, and Dipankar Sarma. Using read-copy update techniques for System V IPC in the Linux 2.5 kernel. In *Proceedings of the 2003 USENIX Annual Technical Conference (FREENIX Track)*, June 2003.

[GKAS99] Ben Gamsa, Orran Krieger, Jonathan Appavoo, and Michael Stumm. Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating system. In *Proceedings of the 3rd Symposium on Operating System Design and Implementation*, New Orleans, LA, February 1999.

[KL80] H. T. Kung and Q. Lehman. Concurrent maintenance of binary search trees. *ACM Transactions on Database Systems*, 5(3):354–382, September 1980.

[LSS02] Hanna Linder, Dipankar Sarma, and Maneesh Soni. Scalability of the directory entry cache. In *Ottawa Linux Symposium*, pages 289–300, June 2002.

[MAK+01] Paul E. McKenney, Jonathan Appavoo, Andi Kleen, Orran Krieger, Rusty Russell, Dipankar Sarma, and Maneesh Soni. Read-copy update. In *Ottawa Linux Symposium*, July 2001.

[McK03] Paul E. McKenney. Using RCU in the Linux 2.5 kernel. *Linux Journal*, 1(114):18–26, October 2003.

[McK04] Paul E. McKenney. *Exploiting Deferred Destruction: An Analysis of Read-Copy-Update Techniques in Operating System Kernels (in preparation).* PhD thesis, Oregon Graduate Institute of Science and Technology, 2004.

[ML84] Udi Manber and Richard E. Ladner. Concurrency control in a dynamic search structure. *ACM Transactions on Database Systems*, 9(3):439–455, September 1984.

[MS98] Paul E. McKenney and John D. Slingwine. Read-copy update: Using execution history

| Configuration | Latency (microseconds) | Standard Deviation |
|---|---|---|
| Base 2.6.0 | 811.0 | 85.26 |
| krcud | 406.4 | 17.87 |
| Direct Invocation | 393.0 | 37.18 |
| Throttle | 414.8 | 37.83 |

Table 1: Realtime Scheduling Latencies


to solve concurrency problems. In *Parallel and Distributed Computing and Systems*, pages 509–518, Las Vegas, NV, October 1998.

[MSA+02] Paul E. McKenney, Dipankar Sarma, Andrea Arcangeli, Andi Kleen, Orran Krieger, and Rusty Russell. Read-copy update. In *Ottawa Linux Symposium*, pages 338–367, June 2002.

[MSS04] Paul E. McKenney, Dipankar Sarma, and Maneesh Soni. Scaling dcache with RCU. *Linux Journal*, 1(118), January 2004.

[Pug90] William Pugh. Concurrent maintenance of skip lists. Technical Report CS-TR-2222.1, Institute of Advanced Computer Science Studies, Department of Computer Science, University of Maryland, College Park, Maryland, June 1990.

[SAH+03] Craig A. N. Soules, Jonathan Appavoo, Kevin Hui, Dilma Da Silva, Gregory R. Ganger, Orran Krieger, Michael Stumm, Robert W. Wisniewski, Marc Auslander, Michal Ostrowski, Bryan Rosenburg, and Jimi Xenidis. System support for online reconfiguration. In *Proceedings of the 2003 USENIX Annual Technical Conference*, June 2003.

[Sar03] Dipankar Sarma. Rcu low latency patches. Message ID: 20031222180114.GA2248@in.ibm.com, December 2003.

[Sar04a] Dipankar Sarma. [patch] rcu for low latency (experimental). `http://marc.theaimsgroup.com/?l=linux-kernel\&m=108003746402892\&w=2`, March 2004.

[Sar04b] Dipankar Sarma. Re: [patch] rcu for low latency (experimental). `http://marc.theaimsgroup.com/?l=linux-kernel\&m=108016474829546\&w=2`, March 2004.

[Sei03] Joseph W. Seigh II. Read copy update. email correspondence, March 2003.

# Indexing Arbitrary Data with SWISH-E

Josh Rabinowitz • joshr@joshr.com
*SkateboardDirectory.com*

**Abstract**

Fast lookups are crucial to many computer applications and operations. The general problem of indexing and searching on arbitrary data is not a simple one, with many semantic, linguistic, and technical issues to iron out. In this paper we present swish-e, a descendent of Kevin Hughes' SWISH project from 1994. Swish-e provides a full-featured and useful toolkit to index and query 8-bit ASCII data. This paper discusses the structure, features, and usage of swish-e, with mentions of possible directions for further development and interesting related work. We also compare swish-e to MySQL's full-text search feature in terms of features and speed, and discuss two real-world swish-e applications, Sman and Swished.

## 1. Introduction

This paper discusses the features and limitations of swish-e[1], and to a lesser extent, MySQL's fulltext search feature[2]. This paper loosely builds on information presented in the Author's article in Linux Journal entitled "How To Index Anything"[3]. We at SkateboardDirectory.com discovered swish-e when researching indexing toolkits and were attracted by its feature set, perl interface, quality documentation, and lively and informative discussion list.

## 2 SWISH-E Overview

The three most common data storage techniques (flat files, Berkeley DB[4]-like binary files, and SQL databases) each give rise to their own particular data search and retrieval features, strengths and weaknesses. While each technique allows some form of easy and/or fast lookups, none is inherently optimized for searching collections of human language text.

Designed not for storage for but quick retrieval of data from prebuilt indices, swish-e fills that need. Swish-e provides a native C interface and command line tools to build and query indices, and a perl interface for searching as well. Indices consist of a pair of binary files and are built using the `swish-e` binary and one of swish-e's three indexing methods. We have been using swish-e at SkateboardDirectory.com since 2002.

## 2.1 Building SWISH-E

Currently, installing swish-e on a unix-like system means building from source. You can find tarballs for the source code on the swish-e website, and swish-e is built through a typical install process using a `./configure` script. The Perl SWISH::API can be found in the /perl subdirectory and is likewise installed through the typical perl `'perl Makefile.PL; make; make test; sudo make install'` process.

## 2.2 Configuration Files

Swish-e will typically depend on a single configuration file while creating a index. These files follow a familiar line-oriented name/value syntax. Blank lines and lines beginning with a # are ignored, remaining lines are expected to be single, named directives. For example, this is a valid swish-e configuration file:

```
# example1.conf
IndexFile example1.index
DefaultContents HTML2
```

## 2.3 SWISH-E Parsers

Swish-e directly supports indexing of text, html, and XML files, converting HTML or XML entities where appropriate, and the ability to index data based on the tags it resides within. The XML2, HTML2, and TXT2 parsing engines, which require the libxml2 libraries, are preferable to the original counterparts (called XML1, HTML1 and TXT1), especially when handling HTML.

## 2.4 Properties and MetaNames

MetaNames are the fields in a swish-e index that are searched on. Properties are the fields returned from a swish-e search describing the particular documents. By default, text is indexed under the MetaName `swishdefault`, and Properties returned indicate information about each relevant document.

Several so-called Auto Properties are always present in search results returned from the swish-e APIs. The table below summarizes some of the most important ones.

Table 1: some of swish-e's Auto Properties

| | |
|---|---|
| swishrank | Normalized integer between 1 and 1000 representing relevance of hit to query |
| swishtitle | Title of the document. Either the filename, or (by default) if parsed from HTML, the text between <title> tags |
| swishdocpath | URL or filepath of indexed document |
| swishdocsize | Length of indexed document, in bytes |

### 2.4.1 String Properties

Any non-numeric properties are internally stored as strings. By default, any properties longer than 100 characters are compressed using zlib before storage, which helps keep the index sizes down. Note that each string is compressed on its own, so redundancy between properties is not exploited in the compression process.

### 2.4.2 Numeric Properties

Using the `PropertyNamesNumeric` directive, swish-e has the ability to store properties as unsigned integers, which allows for proper sorting numerically at search time. Unfortunately, the swish-e engine is not particularly efficient in its methods of sorting by numeric properties: currently the whole result list is simply sorted by integer after the index is searched and before the results are returned to the client.

## 2.5 Indexing Methods

Here we discuss three different ways to index data with swish-e. The first is to index files using one of the built-in parsers alone, as shown in the next section. The other two methods, FileFilters and external programs, allow for conversion of data from other sources or file formats.

### 2.5.1 Indexing Files Directly

Assume that our working directory contains the above `example1.conf` file and a set of HTML files we want to index in ./html, we can build the index `example1.index` using `swish-e`'s `-f` and `-i` options to specify the configuration file to use and which directory to index:

```
swish-e -f example1.conf -i ./html
```

### 2.5.2 Using FileFilters

For data in formats other than HTML, XML or TXT, you need to arrange to have the files converted to one of the formats that swish-e directly supports.

The most straightforward way to convert files for swish-e is via the `FileFilter` method. This is engaged by including lines like the following in your configuration file:

```
# example2.conf
FileFilter .pdf pdftotext "' %p' -"
IndexContents TXT .pdf
```

This specifies that files ending with .pdf are to be converted using the pdftotext executable (part of the xpdf package[5]), and then indexed using swish-e's `TXT` parser. This configuration file would be used similarly to the one shown in the previous example.

The downside to the FileFilter method is that swish-e invokes a child process for each document to be converted. This can present a performance issue during index time. On the other hand, assuming a program exists to perform the translation required, it can be easy to support additional file types by adding a pair of lines like the ones above to your configuration file.

### 2.5.3 Using External Programs

The most flexible mechanism for getting your data into a swish-e index is through External Programs. Essentially, external programs convert documents to a supported format as needed, wrap the result with appropriate swish-e headers, and pass that to `swish-e`. Again, swish-e will only interpret data as TXT, HTML, or XML, so if you have special needs, target your external program's output for a specific one of the parsers.

Here's an example external program that takes all the XML files in the apache 2.0 docs/manual tree, which we've copied to ./manual, and prepares them for indexing with swish-e.

```
#!/usr/bin/perl -w
# example3-prog.pl
# appends data to Path-Name: header

my @files =
`find ./manual -name '*.xml' -print`;

chomp(@files);
my $cnt = 1;
for my $f (@files) {
    open(FILE, "< " . $f);
    my $xml = join("", <FILE>);
    close(FILE);
    my $size = length $xml;
    # note: Fails if UTF
    print "Path-Name: $f $cnt\n",
       "Document-Type: XML*\n",
       "Content-Length: $size\n\n",
        $xml;
    $cnt++;
}
```

We didn't have to use an external program to index the XML files, but doing so allows us to easily introduce how to use some of swish-e's special XML handling features, and to show how to easily add MetaNames and Properties using swish-e's `ExtractPath` and `ReplacePath` directives.

## 2.6 XML, HTML, MetaNames and Properties

Swish-e lets you easily index any text within an HTML or XML tag as a MetaName and/or Property through use of the `MetaNames` and `PropertyNames` directives.

The following configuration file shows use of these directives, along with the `ExtractPath` and `ReplacePath` directives described below:

```
# example3.conf
MetaNames summary docnum swishtitle
PropertyNames summary
PropertyNamesNumeric docnum

# expect path like "/file/path 123"
#act like the 123 was in <docnum> tags
ExtractPath docnum regex \
     '!^.*( [0-9]+)$!$1!'
# remove the " 123" for indexing
ReplaceRules       regex \
      '!^(.*) [0-9]+$!$1!'
```

Note that although our example above uses backslashes to denote continued lines in our configuration file, `swish-e` does not support this feature, so make sure to enter each directive on its own line when writing configuration files.

The `swish-e` executable can be used as shown below to create an index from `example3.conf` and `example3-prog.pl`:

```
swish-e -f example3.index \
 -c example3.conf  \
 -i ./example3-prog.pl \
 -S prog
```

Here, -f sets the path of the index to be created, -c specifies the configuration file to use during the indexing process, and -i sets the program to be used to convert documents for `swish-e`. Lastly, the `-S prog` option denotes that the -i option specifies a program to be executed describing the documents to be indexed. If you forget the `-S prog` option, swish-e will index the file `example3-prog.pl` itself, and not the documents it describes when executed.

### 2.6.1 ExtractPath and ReplacePath

Using `ExtractPath` and `ReplacePath` can be useful for adding meta data to documents to be indexed. In this case, the `ExtractPath` directive serves to let the integer appended to the document path be indexed as though it had appeared inside the document in `<docnum>` tags. The `ReplaceRules` directive then removes the appended integer from the pathname so that it won't appear in the `swishdocpath` when retrieved from a swish-e index.

## 2.7 Searching

There are two main approaches to searching on a swish-e index: using the `swish-e` executable directly, or using one of the APIs to do so.

### 2.7.1 The SWISH-E Query Language

Swish-e supports logical grouping via parenthesis as well as AND, OR and NOT Boolean logic that behave predictably.

### 2.7.2 Searching With SWISH-E

Searching directly with `swish-e` is straightforward but not as flexible as the API. It is nevertheless very valuable for quick tests against indices to see that they work as expected.

For example, we can conduct searches on our `example3.index` like so:

```
swish-e -f example3.index -w restart
```

which returns results like (abridged and reformatted):

```
1000 manual/stopping.XML
      "stopping.XML" 10577
608 manual/platform/windows.XML
      "windows.XML" 30773
608 manual/programs/apachectl.XML
      "apachectl.XML" 5818
544 manual/mod/mpm_common.XML
      "mpm_common.XML" 39171
```

By default, each result contains a rank, the pathname of the indexed file, the title, and the byte count of the indexed data.

### 2.7.3 Using SWISH::API.pm

Here at SkateboardDirectory.com, one of the features that attracted us to swish-e was its perl interface, provided through the included SWISH::API. Most of the important features available for searching through the `swish-e` executable are also accessible through the perl API. Here's a short example that can perform searches similar to the one shown above:

```
#!/usr/bin/perl -w
#search4.pl
use SWISH::API;
my ($max, $cnt) = (10,0);
my $index = "./example3.index";
my $query = join(" ", @ARGV);
my $handle = SWISH::API->new($index);
my $results = $handle->Query($query);
while ( ($cnt++ < $max) &&
(my $res = $results->NextResult)) {
    printf "%d '%s' %d\n",
    $res->Property("swishrank"),
    $res->Property("swishdocpath"),
    $res->Property("swishdocsize");
}
```

Of course, in a real-world application you should probably `use strict` and perform error checking.

### 2.7.4 The SWISH-E C API

There is also a C API for querying swish-e indices. It is similar to the perl API, but allows access to more of the low-level details a swish-e index. For examples on use of the swish-e C API, see the swish-e documentation.

## 2.8 Other Notable SWISH-E Features

Some other features that warrant explanation are described in this section.

### 2.8.1 Merging indices

Swish-e has the ability to search on multiple swish-e indices simultaneously and merge the results meaningfully. This enables searching on groups of indices that may be collectively larger than the current per-index limit of about 2GB.

### 2.8.2 PHP Interface

Many people have expressed interest in a PHP interface to swish-e, and one is in development.[6]

## 2.9 SWISH-E Ranking

The ranking algorithm used in swish–e does not bear easy explanation, but does take into account factors including the size of the documents, the frequency of each word in the document, and which tags the given text resides in. Maintainer Moseley has repeatedly expressed his desire for someone to clean up the ranking code used in swish-e.

# 3 Real-World Examples

Here we examine two real-world uses of swish-e: sman, the Searcher for Man Pages; and swished, a concurrent, persistent swish-e deamon based on mod_perl.

## 3.1 Sman – Searcher For Man Pages

Sman is Searcher for Man Pages (written by the Author) which uses swish–e to offer ranked, fulltext searches on your system's manpages. The Sman package, which is currently available at http://joshr.com/src/sman, is likely to appear on CPAN,

Like most well-designed software, no single part of the sman package is particularly complex. However, due to the wide range of differences in the ways man pages are written, displayed and presented in various software packages and operating systems, there are relatively large amount of moving parts in sman.

As a high-level overview, sman consists of two programs: `sman` and `sman-update`. `Sman` performs the searches on the index of man pages, and `sman-update` upates that index. Sman is broken into a series of perl

modules. By far most of the complexity is employed in `sman-update`.

`Sman-update`, which is intended to be run nightly, does everything necessary to parse your manpages and create a swish-e index which allows freetext searching on the compete text of all the manpages, as well as the ability to search on man pages by text in their command, section, pathname, or description. In a little more detail, `sman-update`:

- reads a configuration file (by default /usr/local/etc/sman-default.conf) which specifies options including where to store the final index, (by default /var/lib/sman/sman.index)
- finds your manpages
- figures out how to best convert your man pages to ASCII
- creates a temporary swish-e configuration file for use while indexing the man pages
- converts each manpage and parses the result to ascertain the title, description, section, and complete text.
- outputs XML to swish-e to parse, using the temporary configuration file

Sman, the tool that actually performs searches on the index, is essentially a highly enhanced version of `search4.pl`, shown above. We've skipped the details of the object-oriented design used. For more details, see the Sman source code and documentation.[7]

## 3.2 swished – A SWISH-E Daemon

For some time, one of the items on the swish-e to-do list has been for someone to write a persistent, concurrent server for swish-e indices. Here we present the overall design of such a daemon, written as an apache mod_perl handler.

There are at least three reasons that apache makes sense as an infrastructure for this purpose.

- http is a well defined protocol
- apache is stable, proven, widely deployable software that provides sophisticated logging, authentication, and extension mechanisms
- Perl, SWISH::API and mod_perl make it fairly easy

The plan is to provide a SWISH::API::Remote perl module that will access a swished daemon using an interface very similar to SWISH::API, but which is designed to communicate with swished over TCP/IP instead of directly reading the swish-e index.

# 4 SWISH-E vs MySQL

Just how fast is swish-e? Here we put swish-e and MySQL's fulltext search feature through the paces with some benchmarks against indices of different sizes and compare the results.

## 4.1 Differences and Similarities

While targeted at essentially the same problem of facilitating quick searches on larger amounts of textual data, the indexing models employed by MySQL's Full Text search and swish-e have significant differences. Some of these are outlined below.

Table 2: Some Pros and Cons of swish-e and MySQL

| Engine | Pros | Cons |
|---|---|---|
| Swish-e | • more compact indices<br>• indexes all words over 1 character | • not multibyte<br>• no updates to indices*<br>• only for indexing |
| MySQL | • updatable indices*<br>• deep multibyte support<br>• also a storage engine | • less compact indices<br>• indexes only words over 3 characters**<br>• ignores words appearing in over 50% of rows in a fulltext index*** |

* In swish-e, indexes must be rewritten to be modified. MySQL indices can be updated through normal sql queries.
** By default, MySQL will only index words four charaters or longer. This can be changed via MySQL's `ft_min_word_len` configuration option.
*** MySQL fulltext search does consider such words if searches are conducted in `boolean` mode, but then the results aren't ranked.

## 4.2 Benchmark Methodology

To compare swish-e with MySQL, a series of textual collections were created, varying in size from 1MB to 5GB. Each corpus was made of English words randomly chosen but based on the statistical

frequencies of common English words. The following table summarizes the seven collections used in the benchmarks:

Table 3: breakdown of collections used for testing.

| Approx Collection Size | Num Words/ Doc | Num Docs | Approx size of swish-e index | Approx size of MySQL index |
|---|---|---|---|---|
| 1MB | 15 | 8.3K | 2.6MB | 2.3MB |
| 3MB | 15 | 25K | 5.9MB | 6.8MB |
| 10MB | 54 | 25K | 14MB | 21MB |
| 50MB | 54 | 100K | 56MB | 109MB |
| 250MB | 273 | 100K | 245MB | 554MB |
| 1GB | 273 | 500K | 970MB | 2.0GB |
| 5GB | 1366 | 500K | 4.6GB* | 9.0GB |

*The swish-e index for the 5GB collection was built in two equal-sized parts, as swish-e cannot yet support index files over about 2.1GB. The two indices used in the 5GB collection were searched simultaneously using swish-e's merge search feature.

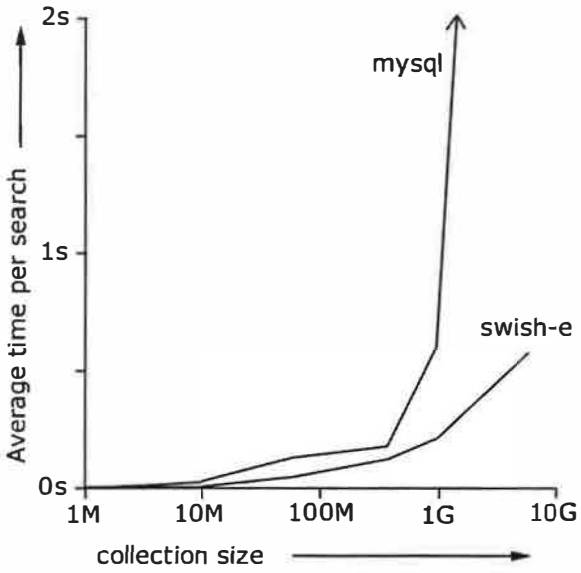
Tests were performed on a 3.06Ghz Hyper-Threading Pentium 4 with 2GB of RAM and an 80GB HD. The system was running a linux 2.6.5 kernel with SMP enabled. Swish-e version 2.5.1-2004-04-28 and MySQL 4.1.1 were used.

For MySQL, the provided `my-huge.cnf` configuration file was used, with the following modifications: `ft_min_word_len` was set to three instead of four, thus indexing words three or more letters (and not only words four letters or longer), and thread_concurrency was set to 4. Additionally, for the 1GB and 5GB indices, the MySQL configuration option MAX_ROWS was set to 10000000 and AVG_ROW_LENGTH was set to 13000 so as to allow enough data to be stored in each table.

The searches were performed with a two groups of 200 searches made up of randomly selected words from the collections three characters or longer. In each group of 200 searches, 53 were 1-word searches, 115 were 2-word searches, 25 were 3-word searches, 4 were 4-word searches, 2 were 5-word searches, and 1 was a six-word search. (This approximates the distribution of numbers of words in the most popular searches on SkateboardDirectory.com.) All swish-e searches were conducted with the words ANDed, and MySQL searches were conducted with the + prefix, which has the effect of requiring relevant documents to contain all words in the query. Each index was tested by searching twice with two sets of 200 queries, retrieving the pathname and complete content of the first 20 documents found relevant, and computing the average response time. The MySQL server was not running when swish-e was being benchmarked, and the machine was rebooted between each test run.

## 4.3 Benchmark Results

Graph 1: MySQL against swish-e, showing average search time vs. collection size.




(The average search time for searches on the 5GB MySQL collection was 14 seconds, way off the graph above.)

As the graph above indicates, both engines are very fast, but swish-e is faster, especially for large indices. When searching on the 5GB collection using MySQL, it appears that performance suffers as the machine begins to thrash, as the index can no longer fit in memory and/or MySQL's caches. Adjusting MySQL's configuration may help narrow this gap but it appears that for now, swish-e is faster for raw searches. There is also likely more speed to be squeezed out of swish-e.

## 5 Research Ideas

As alluded to before in section 2.9, maintainer Moseley thinks that work on the ranking algorithm in swish-e is worthy of a graduate level thesis in computer science. Considering that Google is based on a 1998 Stanford hypertext retrieval project focused on ranking web pages[8], this is probably an understatement.

## 6 Limitations and Weaknesses

For all its strengths, swish-e has some weaknesses.

### 6.1 Size limits

There are various hard-coded and system-oriented size limits in swish-e, including limits on the maximum lengths of words, Properties, and indices as a whole.

### 6.2 Character Sets And Conversion Issues

Also, swish-e is admittedly an 8bit indexing system, and has no multibyte nor UTF support. Indices must be built and queried for a particular character set.

### 6.3 Occasionally Quirky Search Results

The ranking algorithm of swish-e occasionally leads to surprising, even erroneous search results, typically by failing to rank highly documents that one would expect to be relevant.

## 7 Future Plans for SWISH-E

These are some features the developers of swish-e feel are important.

### 7.1 UTF-8 support

This would enable a single index to hold data in languages that require multibyte characters, and would make swish-e indexes fully 8bit aware. According to Moseley, adding support for UTF-8, which he believes to be the best course of action, would require a near total rewrite of swish-e because of many assumptions that the size of one character is one byte.

### 7.2 Remove Two Gigabyte Limits

The files used in a swish-e index are currently limited to about 2GB in size, even on systems which support larger files. Support for indices larger than 2GB is in development.

### 7.3 Ranking Improvements

Mentioned above, there is room for improvement in the ranking algorithms.


## 8.0 Acknowledgements

Thanks to Kevin Hughes, Bill Moseley, Jóse Manuel Ruiz, David Norris, Roy Tennant, and Adam Souzis for their input and feedback on this paper. In addition, thanks to the entire current and past swish, swish-e, and MySQL development teams for great open source software products.


## 9.0 Conclusion

It is clear that swish-e is a powerful and fast engine with which to create and search on indices. The underlying speed of swish-e, coupled with its quality documentation, lively discussion list, perl, C, and PHP interfaces, provide a robust and flexible foundation upon which to build searching systems.

## References


[1] Hughes, K., Moseley, B. et. Al.: *SWISH-E.* `www.swish-e.org`, 2004.

[2] MySQL AB: *MySQL.* `www.mysql.com`, 2004.

[3] Rabinowitz, J.: *How To Index Anything*; Linux Journal, July 2003, also at `www.linuxjournal.com/article.php?sid=6652`

[4] Sleepycat Software: *Berkeley DB.* `www.sleepycat.com`, 2004.

[5] Glyph & Cog, *Xpdf*; `www.foolabs.com/xpdf/`

[6] Ruiz, J. M., *php-swishe*; `http://prdownloads.sourceforge.net/php-swishe/`

[7] Rabinowitz, J.: *Sman: The Searcher for Man Pages;* `www.joshr.com/src/sman/`

[8] Page, L., Brin, S., Motwani, R., Winograd, T.: *The PageRank Citation Ranking: Bringing Order to the Web*; Stanford Digital Libraries Project, 1998.

# Towards Carrier Grade Linux Platforms

Ibrahim Haddad
Ericsson Research
8400 Decarie Blvd, Montreal
Quebec H4P 2N2, Canada
*ibrahim.haddad@ericsson.com*

## Abstract

Traditionally, communications and data service networks were built on proprietary platforms that had to meet very specific availability, reliability, performance, and service response time requirements. Today, communications service providers are challenged to meet their needs cost-effectively for new architectures, new services, and increased bandwidth, with highly available, scalable, secure, and reliable systems that have predictable performance and that are easy to maintain and upgrade. This paper presents the technological trend of migrating from proprietary to open platforms based on software and hardware building blocks. The paper focuses on the ongoing work by the Carrier Grade Linux (CGL) working group at the Open Source Development Labs, examines the CGL architecture, the requirements from the latest specification release, and presents some of the needed kernel features that are not currently supported on Linux.

## 1. Open, standardized, and modular platforms

The demand for rich media and enhanced communications services is rapidly leading to significant changes in the communications industry such as the convergence of data and voice technologies. The transition to packet-based, converged, multi-service IP networks require a carrier grade infrastructure based on interoperable hardware and software building blocks, management middleware and applications, implemented with standard interfaces.

The communications industry is witnessing a technology trend moving away from proprietary systems toward open and standardized systems, built using modular and flexible hardware and software (operating system and middleware) common off the shelf components. The trend is to proceed forward delivering next generation and multimedia communication services, using open standard carrier grade platforms. This trend is motivated by the expectations that open platforms are going to reduce the cost and risks of developing and delivering rich communications services; they will enable faster time to market and ensure portability and interoperability between various components from different providers.

One frequently asked question is: How can we meet tomorrow's requirements using existing infrastructures and technologies? Proprietary platforms are closed systems, expensive to develop, and often lacking support of the current and upcoming standards. Using such *closed* platforms to meet tomorrow's requirements for new architectures and services is almost impossible. A uniform, open software environment with the characteristics demanded by telecom applications, combined with commercial off-the-shelf software and hardware components is a necessary part of these new architectures.

Three key industry consortia are defining hardware and software high availability specifications that are directly related to telecom platforms:

- The PCI Industrial Computer Manufacturers Group [1] (PICMG) defines standards for high availability (HA) hardware.
- The Open Source Development Labs [2] (OSDL) Carrier Grade Linux [3] (CGL) working group was established in January 2002 with the goal of enhancing the Linux operating system, to achieve an Open Source platform that is highly available, secure, scalable and easily maintained, suitable for carrier grade systems.
- The Service Availability Forum [4] (SA Forum) defines the interfaces of HA middleware and focusing on APIs for hardware platform management and for application failover in the application API. SA compliant middleware will provide services to

an application that needs to be HA in a portable way.

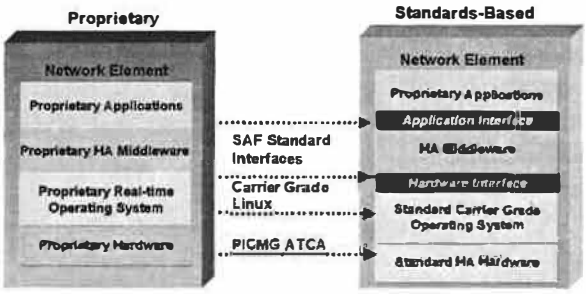



**Figure 1: From Proprietary to Open Solutions**

The operating system is a core component in such architectures. In the remaining of this paper, we will be focusing on CGL, its architecture and specifications.

## 2. The term Carrier Grade

In this paper, we refer to the term Carrier Grade on many occasions. Carrier grade is a term for public network telecommunications products that require a reliability percentage up to 5 or 6 nines of uptime.
5 nines of uptime refer to 99.999% of uptime (i.e. 5 minutes of downtime per year). This level of availability is usually associated with Carrier Grade servers.
6 nines of uptime refer to 99.9999% of uptime (i.e. 30 seconds of downtime per year). This level of availability is usually associated with Carrier Grade switches.

## 3. Linux versus proprietary operating systems

This section describes briefly the motivating reasons in favor of using Linux on Carrier Grade systems, versus continuing with proprietary operating systems. These motivations include:

- Cost: Linux is available free of charge in the form of a downloadable package from the Internet.
- Source code availability: With Linux, you gain full access to the source code allowing you to tailor the kernel to your needs.
- Open development process (Figure 2): The development process of the kernel is open to anyone to participate and contribute. The process is based on the concept of "release early, release often."
- Peer review and testing resources: With access to the source code, people using a wide variety of platform, operating systems, and compiler combinations; can compile, link, and run the code on their systems to test for portability, compatibility and bugs.
- Vendor independent: With Linux, you no longer have to be locked into a specific vendor. Linux is supported on multiple platforms.
- High innovation rate: New features are usually implemented on Linux before they are available on commercial or proprietary systems.

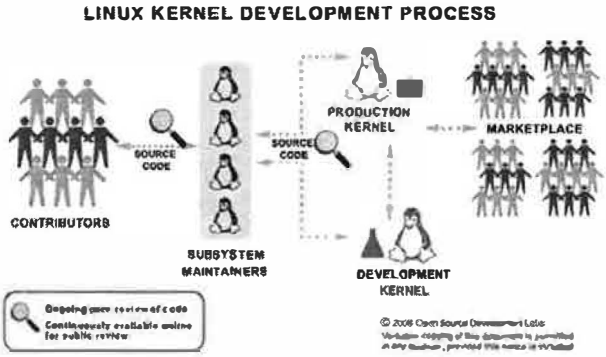



**Figure 2: Open development process of the Linux kernel**

Other contributing factors include Linux' support for a broad range of processors and peripherals, commercial support availability, high performance networking, and the proven record of being a stable, and reliable server platform.

## 4. Carrier Grade Linux

The Linux kernel is missing several features that are needed in a telecom environment, and it is not adapted to meet telecom requirements in various areas such as reliability, security, and scalability. To help the advancement of Linux in the telecom space, OSDL established the CGL working group. The group specifies and help implement an Open Source platform targeted for the communication industry that is highly available, secure, scalable and easily maintained, suitable for carrier grade systems.
The CGL working group is composed of several members from network equipment providers, system integrators, platform providers, and Linux distributors, all of them contributing to the requirement definition of Carrier Grade Linux, helping Open Source projects to meet these requirements, and in some cases starting new Open Source projects. Many of the CGL members companies have contributed pieces of technologies to Open Source in order to make the Linux Kernel a more viable option for telecom platforms. For instance, the Open Systems Lab [5] from Ericsson Research has contributed three key technologies: the Transparent IPC [6], the Asynchronous Event Mechanism [7], and the Distributed Security Infrastructure [8]. In a different

direction, there are already Linux distributions, MontaVista [10] for instance, that are providing CGL distribution based on the CGL requirement definitions. Many companies are also either deploying CGL, or at least evaluating and experimenting with it.

Consequently, CGL activities are giving much momentum for Linux in the telecom space allowing it to be a viable option to proprietary operating system. Member companies of CGL are releasing code to Open Source and making some of their proprietary technologies open, going forward from closed platforms to open platforms that use CGL.

## 5. Target CGL applications

The CGL Working Group has identified three main categories of application areas into which they expect the majority of applications implemented on CGL platforms to fall. These application areas are *gateways*, *signaling,* and *management servers.*

- *Gateways* are bridges between two different technologies or administration domains. For example, a media gateway performs the critical function of converting voice messages from a native telecommunications time-division-multiplexed network, to an Internet protocol packet-switched network. A gateway processes a large number of small messages received and transmitted over a large number of physical interfaces. Gateways perform in a timely manner very close to hard real-time. They are implemented on dedicated platforms with replicated (rather than clustered) systems used for redundancy.

- *Signaling servers* handle call control, session control, and radio recourse control. A signaling server handles the routing and maintains the status of calls over the network. It takes the request of user agents who want to connect to other user agents and routes it to the appropriate signaling. Signaling servers require soft real time response capabilities less than 80 milliseconds, and may manage tens of thousands of simultaneous connections. A signaling server application is context switch and memory intensive due to requirements for quick switching and a capacity to manage large numbers of connections.

- *Management servers* handle traditional network management operations, as well as service and customer management. These servers provide services such as: a Home Location Register and Visitor Location Register (for wireless networks) or customer information (such as personal preferences including features the customer is authorized to use). Typically, management applications are data and communication intensive. Their response time requirements are less stringent by several orders of magnitude, compared to those of signaling and gateway applications.

## 6. Overview of the CGL working group

The CGL working group has the vision that next-generation and multimedia communication services can be delivered using Linux based open standards platforms for carrier grade infrastructure equipment. To achieve this vision, the working group has setup a strategy to define the requirements and architecture for the Carrier Grade Linux platform and develop a roadmap for the platform and to promote development of a stable platform upon which commercial components and services can be deployed.

In the course of achieving this strategy, the OSDL CGL working group, is creating the requirement definitions, and identifying existing Open Source projects that support the roadmap to implement the required components and interfaces of the platform. When an Open Source project does not exist to support a certain requirement, OSDL CGL is launching (or support the launch of) new Open Source projects to implement missing components and interfaces of the platform.

The CGL working group consists of three distinct sub-groups that work together. These sub-groups are: specification, proof-of-concept, and validation. Explanations of the responsibilities of each sub-group are as follows:

Specifications: The specifications sub-group is responsible for defining a set of requirements that lead to enhancements in the Linux kernel, that are useful for carrier grade implementations and applications. The group collects, categorizes, and prioritizes the requirements from participants to allow reasonable work to proceed on implementations. The group also interacts with other standard defining bodies, open source communities, developers and distributions to ensure that the requirements identify useful enhancements in such a way, that they can be adopted into the base Linux kernel.

- Proof-of-Concept: This sub-group generates documents covering the design, features, and technology relevant to CGL. It drives the implementation and integration of core Carrier Grade enhancements to Linux as identified and prioritized by the requirement document. The group is also responsi-

ble for ensuring the integrated enhancements pass, the CGL validation test suite and for establishing and leading an open source umbrella project to coordinate implementation and integration activities for CGL enhancements.

- Validation: The sub-group defines standard test environments for developing validation suites. It is responsible for coordinating the development of validation suites, to ensure that all of the CGL requirements are covered. This group is also responsible for the development of an Open Source project CGL validation suite.

## 7. CGL architecture

Figure 3 presents the scope of the CGL Working Group, which covers two areas:

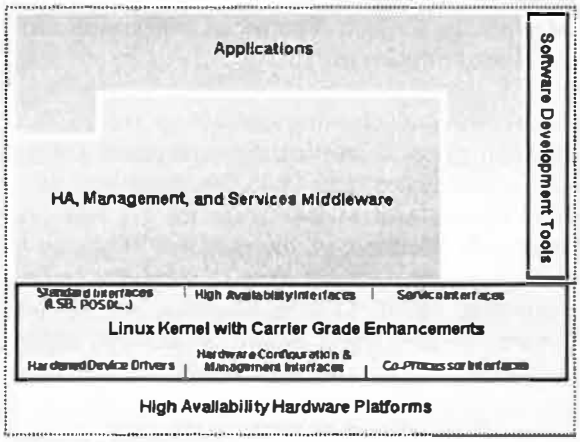



**Figure 3: CGL architecture and scope**

1. *Carrier Grade Linux*: Various requirements such as availability and scalability are related to the CGL enhancements to the operating system. Enhancements may also be made to hardware interfaces, interfaces to the user level or application code and interfaces to development and debugging tools. In some cases, to access the kernel services, user level library changes will be needed.

2. *Software Development Tools*: These tools will include debuggers and analyzers.

On October 9, 2003, OSDL announced the availability of the OSDL Carrier Grade Linux Requirements Definition, Version 2.0 (CGL 2.0). This latest requirement definition for next-generation carrier grade Linux offers major advances in security, high availability, and clustering.

## 8. CGL 2.0 requirements

The requirement definition document of CGL version 2.0 introduced new and enhanced features to support Linux as a carrier grade platform. The CGL requirement definition divides the requirements in main categories described briefly below:

### 8.1 Clustering

These requirements support the use of multiple carrier server systems to provide higher levels of service availability through redundant resources and recovery capabilities, and to provide a horizontally scaled environment supporting increased throughput.

### 8.2 Security

The security requirements are aimed at maintaining a certain level of security while not endangering the goals of high availability, performance, and scalability. The requirements support the use of additional security mechanisms to protect the systems against attacks from both the Internet and intranets, and provide special mechanisms at kernel level to be used by telecom applications.

### 8.3 Standards

CGL specifies standards that are required for compliance for carrier grade server systems.
Examples of these standards include:

- Linux Standard Base
- POSIX Timer Interface
- POSIX Signal Interface
- POSIX Message Queue Interface
- POSIX Semaphore Interface
- IPv6 RFCs compliance
- IPsecv6 RFCs compliance
- MIPv6 RFCs compliance
- SNMP support
- POSIX threads

### 8.4 Platform

OSDL CGL specifies requirements that support interactions with the hardware platforms making up carrier server systems. Platform capabilities are not tied to a particular vendor's implementation.
Examples of the platform requirements include:

- Hot insert: supports hot-swap insertion of hardware components.
- Hot remove: supports hot-swap removal of hardware components.
- Remote boot support: supports remote booting functionality.
- Boot cycle detection: supports detecting reboot cycles due to recurring failures.

- Diskless systems: support diskless systems which load and run applications via the network.

### 8.5 Availability

The availability requirements support heightened availability of carrier server systems, such as improving the robustness of software components or by supporting recovery from failure of hardware or software.
Examples of these requirements include:

- RAID 1: support for RAID 1 offers mirroring to provide duplicate sets of all data on separate hard disks.
- Watchdog timer interface: support for watchdog timers to perform certain specified operations when timeouts occur.
- Support for Disk and volume management: to allow grouping of disks into volumes.
- Ethernet link aggregation and link failover: support bonding of multiple NIC for bandwidth aggregation and provide automatic failover of IP addresses from one interface to another.
- Support for application heartbeat monitor: monitor applications availability and functionality.

### 8.6 Serviceability

The serviceability requirements support servicing and managing hardware and software on carrier server systems. These are wide-ranging set requirements that, put together, help support the availability of applications and the operating system.
Examples of these requirements include:

- Support for producing and storing kernel dumps.
- Support for dynamic debug of the kernel and running applications.
- Support for platform signal handler enabling infrastructures to allow interrupts generated by hardware errors to be logged using the event logging mechanism.
- Support for remote access to event log information.

### 8.7 Performance

OSDL CGL specifies the requirements that support performance levels necessary for the environments expected to be encountered by carrier server systems.
Examples of these requirements include:

- Support for application (pre) loading.
- Support for soft real time performance through configuring the scheduler to provide soft real time support with latency of 10 ms.
- Support Kernel preemption.
- Provide Raid 0 support to enhance performance.

### 8.8 Scalability

These requirements support vertical and horizontal scaling of carrier server systems such as the addition of hardware resources to result in acceptable increases in capacity.

### 8.9 Tools

The tools requirements provide capabilities to facilitate diagnosis. Examples of these requirements include:

- Support the usage of a kernel debugger.
- Support for Kernel dump analysis.
- Support for debugging multi-threaded programs

## 9. CGL 3.0

The work on the next version of the OSDL CGL requirements, version 3.0, started in January 2004 with focus on advanced requirement areas such as manageability, serviceability, tools, security, standards, performance, hardware, clustering and availability. With the success of CGL's first two requirement documents, OSDL CGL working group anticipate that their third version will be quite beneficial to the Carrier Grade ecosystem. Official release of the CGL requirement document Version 3.0 is expected in October 2004.

## 10. CGL implementations

There are several enhancements to the Linux Kernel that are required by the communication industry, to help adopt Linux on their carrier grade platforms, and support telecom applications. These enhancements (Figure 4) fall into the following categories availability, security, serviceability, performance, scalability, reliability, standards, and clustering.

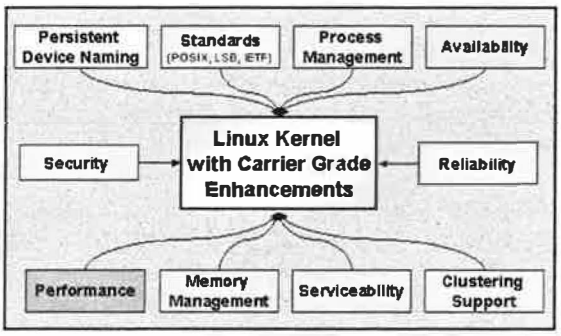



**Figure 4: CGL enhancements areas**

The implementations providing theses enhancements are Open Source projects and planned for integration with the Linux kernel when the implementations are mature, and ready for merging with the kernel code. In some cases, bringing some projects into maturity levels takes a considerable amount of time before being able to request its integration into the Linux kernel. Never-

theless, some of the enhancements are targeted for inclusion in kernel version 2.7. Other enhancement will follow in later kernel releases. Meanwhile, all enhancements, in the form of packages, kernel modules and patches, are available from their respective project web sites.
The CGL 2.0 requirements are in-line with the Linux development community. The purpose of this project is to form a catalyst to capture common requirements from end-users for a CGL distribution. With a common set of requirements from the major Network Equipment Providers, developers can be much more productive and efficient within development projects. Many individuals within the CGL initiative are also active participants and contributors in the Open Source development community

In this section, we provide some examples of missing features and mechanisms from the Linux kernel that are necessary in a telecom environment.

Linux should be able to run in a routing environment with fast recovery of routes when network failure is detected. This is achievable by having around 2000 routes/sec. Latency is not really an issue in a PC environment (a few ms doesn't make a big difference). What is important is to have a predictable performance from 10.000 to 500.000 routes. However, the faster is always better.
The actual implementation of the IP stack in Linux works fine for host or small router. However, with the high requirements in telecom, it becomes impossible to develop using Linux a high-end router for large network (core/border/access router) or a high-end server with routing capabilities.
The problem we are facing with Linux is the lack of support for multiple forwarding information bases (multi-FIB) with overlapping interface's IP address and appropriate interfaces for addressing FIB(VR). The route table is not scalable.
Another objective is to support multi-FIB with overlapping IP address. We can have on different VLAN or different physical interface, independent network in the same Linux box. For example, you can have 2 HTTP servers serving 2 different networks with potentially the same IP address. One HTTP serves the network/FIB 10, and the other serves the network/FIB 20. So the advantage you have is to have 1 Linux box serving 2 different customers with the own networks. (i.e. ISP with big companies using there services). So the only way to achieve that is to have an ID to completely separate the table in memory. (i.e. can be separate table or the ID is just append at the beginning of the key).
Another problem arise when we are not able to predict access time, with the chaining in the hash table of the routing cache (and FIB). This problem is of particular interest in environment that requires predictable performance.
Another aspect of the problem is that the route cache and the routing table are not kept synchronized most of the time (path MTU, just to name one). The route cache flush is executed regularly therefore any updates on the cache are lost. For example, if you have a routing cache flush, you have to rebuild every route that you are currently talking to. To achieve that, you need to go for every route in the hash/try table and rebuild the information. So you first have to lookup in the routing cache, if you have a miss, you need to go in the hash/try table. It's a very slow and not predictable because in the hash/try table with linked list with also a lot of potential collision when a large number of routes are present. This design is perfect for a home PC with a few routes, but it is not scalable for a large server.

To support the various routing requirements of telecom platforms, Linux should support:
- Implementation of multi-FIB using tree (radix, patricia, etc.). It is very important to have predictable performance in insert/delete/lookup from 10 to 500k routes. And, if possible, to have the same data structure for both IPv4 and IPv6.
- Socket and ioctl interfaces for addressing multi-FIB, and
- Multi-FIB support for neighbors (arp)

Providing these implantations in Linux will affect a large part of net/core, net/ipv4 and net/ipv6; these subsystems (mostly network layer) will need to be rewritten. Other areas will have minimal impact at the source code level, mostly at the transport layer (socket, TCP, UDP, RAW, NAT, IPIP, IGMP, etc).
There is no Open Source solutions or patches that are available.

Operating systems for telecom applications must ensure that they can deliver a high response rate with minimum downtime, less than five minutes per year of downtime, including hardware, operating system and software upgrade. In addition to this goal, a carrier-grade system also must take into account such charac-

teristics as scalability, high availability and performance.
For such systems, thousands of requests must be handled concurrently without affecting the overall system's performance, even under extremely high loads. Subscribers can expect some latency time when issuing a request, but they are not willing to accept an unbounded response time. Such transactions are not handled instantaneously for many reasons, and it can take some milliseconds or seconds to reply. Waiting for an answer reduces applications' abilities to handle other transactions.
Many different solutions have been envisaged to improve Linux's capabilities using different types of software organization, such as multithreaded architectures, by implementing efficient POSIX interfaces or by improving the scalability of existing kernel routines. We think that none of these solutions are adequate for true Carrier Grade servers.
As a result, Ericsson has designed and developed the needed mechanism for telecom application and released it to Open Source under the GPL license. The solution is called Asynchronous Event Mechanism (AEM); it provides asynchronous execution of processes in the Linux kernel. AEM implements a native support for asynchronous events in the Linux kernel and aims to bring carrier-grade characteristics to Linux in areas of scalability and soft real-time responsiveness. In addition, AEM offers event-based development framework, scalability, flexibility and extensibility.
AEM has been announced on the Linux Kernel Mailing List (LKML) and received feedback that resulted in some changes to the design and implementation. AEM is not yet integrated with the Linux kernel. More information on AEM is available from [7].

### 11.3 Transparent inter-process and inter-processor communication protocol

Today's telecommunication environments are increasingly adopting clustered servers to gain benefits in performance, availability, and scalability. The resulting benefits of a cluster are greater or more cost-efficient than what a single server can provide. Furthermore, the telecommunication industry's interest in clustering originates from the fact that clusters ad-dress carrier-class characteristics such as guaranteed service availability, reliability and scaled performance, using cost-effective hardware and software. Without being absolute about these requirements, they can be divided in these three categories: short failure detection and failure recovery, guaranteed availability of service, and short response times.
The most widely adopted clustering technique is use of multiple interconnected loosely coupled nodes to a single highly available system.
One missing feature from Linux in this area is a reliable and efficient inter-process and inter-processor communication protocol. However, there exist an Open Source project, Trans-parent Inter Process Communication (TIPC) protocol, which is specially designed for efficient intra cluster communication, leveraging the particular conditions present within loosely coupled clusters. It runs on Linux and is provided as a portable source code package implementing a loadable kernel module.
TIPC is unique from the perspective that there seems to be no other protocol providing a comparable combination of versatility and performance. The functional addressing scheme is an original innovation, as is the topology subscription services and its "reactive connection" concept. TIPC is a useful toolbox for anyone wanting to develop or use Carrier Grade or Highly Available clusters on Linux. It provides the necessary infrastructure for cluster, network and software management functionality, as well as a good support for designing site-independent, scalable, distributed, high-availability and high-performance applications.
Some of the most important TIPC features include full location transparency, lightweight connections, reliable multicast, signaling link protocol, topology subscription services and more.
TIPC is a contribution from Ericsson to Open Source. It will be announced to LKML in mind-May 2004, two weeks after I submit the paper the USENIX. However, more recent news regarding TIPC will be included in the USENIX presentation. TIPC is licensed under a dual GPL and BSD license. More information on TIPC is available from [6][11].

### 11.4 Run-time authenticity verification for system binaries

Linux has generally been considered immune to the spread of viruses, backdoors and Trojan programs on the Internet. However, with the increasing popularity of Linux as a desktop platform, the risk of seeing viruses or Trojans developed for this platform are rapidly growing. To alleviate this problem, the system should prevent on run time the execution of untrusted software. One solution is to digitally sign the trusted binaries and have the system check the digital signature of binaries before running them. Therefore, untrusted (not signed) binaries are denied the execution. This can improve the security of the system by avoiding a wide range of malicious binaries like viruses, worms, Torjan programs and backdoors from running on the system.
DigSig Linux kernel module checks the signature of a binary before running it [9][12]. It inserts digital signatures inside the ELF binary and verifies this signature before loading the binary. It is based on the Linux Se-

curity Module hooks (main stream Linux kernel from 2.5.X and higher).

Typically, in this approach, vendors do not sign binaries; the control of the system remains with the local administrator. S/he is responsible to sign all binaries s/he trusts with his/her private key. Therefore, DigSig guarantees two things: (1) if you signed a binary, nobody else than you can modify that binary without being detected, and (2) nobody can run a binary which is not signed or badly signed.

There have already been several initiatives in this domain, such as Tripwire, BSign, Cryptomark [14][15][16]. We believe the DigSig project is the first to be both easily accessible to all (available on SourceForge, under the GPL license), and it operates at kernel level on run time. The run time is very important for CGL as this takes into account the high availability aspects of the system.

The DigSig approach has been to use the existing solutions like GPG [13] and BSign [15] (a Debian package) rather than reinventing the wheel. However, in order to reduce the overhead in the kernel, the DigSig project only took the minimum code necessary from GPG. This helped much to reduce the amount of code imported to the kernel in source code of the original (only 1/10 of the original GnuPG 1.2.2 source code has been imported to the kernel module).

## 12. Conclusion

There are many challenges accompanying the migration from proprietary to open platforms. The main challenge remains to be the availability of the various kernel features and mechanisms needed for telecom platforms and integrating these features in the Linux kernel.

Carrier Grade Linux is a cooperative initiative aiming to advance the Linux in the communications space and provide an alternative away from proprietary carrier grade operating systems. The participation in OSDL CGL is open to everyone. For more information, please visit the OSDL web site.

## References


[1] PCI Industrial Computer Manufacturers Group, http://www.picmg.org

[2] Open Source Development Labs, http://www.osdl.org

[3] Carrier Grade Linux, http://osdl.org/lab_activities/carrier_grade_linux

[4] Service Availability Forum, http://www.saforum.org

[5] Open System Lab, http://www.linux.ericsson.ca

[6] Transparent IPC, http://tipc.sf.net

[7] Asynchronous Event Mechanism, http://aem.sf.net

[8] An Event Mechanism for Linux, Linux Journal, July 2004, http://www.linuxjournal.com/print.php?sid=6777

[9] Distributed Security Infrastructure, http://disec.sf.net

[10] MontaVista Carrier Grade Edition http://www.mvista.com/cge/index.html

[11] Make Clustering Easy with TIPC, LinuxWorld Magazine, April 2004, http://www.linux.ericsson.ca/papers/tipc_lwm/

[12] Stop Malicious Code Execution at Kernel Level, LinuxWorld Magazine, January 2004, http://www.linux.ericsson.ca/papers/digsig_lwm.pdf

[13] GnuPG, http:// www.gnupg.org

[14] Tripwire, http://www.tripwire.com

[15] Bsign, http://packages.qa.debian.org/b/bsign.html

[16] Cryptomark, http://www.immunix.org/cryptomark.html

# Extreme Linux SIG Session

# Cluster Interconnect Overview

Brett M. Bode, Jason J. Hill, and Troy R. Benjegerdes
*Scalable Computing Laboratory*
*Ames Laboratory*
*Ames, Iowa 50011*

## Abstract

Today cluster computers are more commonplace than ever and there are a variety of choices for the interconnect. The right choice for a particular installation will depend on a variety of factors including price, raw performance, scalability, etc. This paper will present an overview of the popular network technologies available today including Gigabit Ethernet, 10 Gigabit Ethernet, Myrinet, SCI, Quadrics, and InfiniBand. Where possible a comparison will be included for multiple vendors of a given technology. Included will be comparisons of cost and performance of each along with suggestions for when each might present the best choice for a cluster installation.

## 1. Introduction

Over the past few years the cost and performance of interconnects has progressed to the point where today most new clusters use a primary interconnect of 1 – 10 Gbps. There are several interconnection choices in this performance range that range in cost, latency and achievable bandwidth. Choosing the correct one for a particular application is an important and often expensive decision. This paper will present a direct comparison of Gigabit Ethernet, 10 Gigabit Ethernet, Myrinet, SCI, Quadrics and InfiniBand. For each of the network technologies we will examine issues of cost, performance (latency and bandwidth), and scalability.

Some of the network interconnects in this review have been around for quite some time such as Gigabit Ethernet and Myrinet. Others such as InfiniBand and 10 Gigabit Ethernet are quite new. In addition even well known technologies such as Myrinet are evolving in terms of both hardware and software implementations. For example Myricom now offers both single and dual port NICs and is in the process of finishing a substantial rewrite of their software stack.

Finally we will examine application behavior with each of the interconnect technologies. Even the highest performing network will be of little use if problems in the software stack cause scalability problems for real applications.

## 2. Point to Point Performance

The most important parameters for an interconnect are the latency and bandwidth that an application would experience. To measure these parameters we have used the NetPipe[1] program running between two Dell 2650's with dual 2.4Ghz Pentium 4 CPUs with the NICs installed in the PCIX 133Mhz slot. NetPipe functions as a normal user application using either MPI or TCP interfaces. NetPipe can also be used directly with low level interfaces, but since that is less relevant to applications we will not consider it in this work.

NetPipe measures performance versus message size over an exponentially increasing message size. This data is then plotted on a semi-log plot of bandwidth versus message size. There are several common traits to these graphs. First off the bandwidth achieved for small messages is very low for any type of interconnect since it is latency dominated. In addition most interconnects only achieve peak performance for messages over 128KB, some not until messages over 1MB. Finally message sizes in the range of 10KB - 1 MB tend to be the most relevant to applications.

### 2.1. Gigabit Ethernet

Ethernet of some sort has always been used in cluster computers and even today it is present in almost all clusters. While Fast Ethernet was often used as the high-speed interconnect in years past, it has been relegated to the service network today in most systems. The primary limiting factor with Ethernet in the cluster world has always been the switch. Since the underlying architecture of Ethernet requires smart switches which shoulder the full burden of routing packets Ethernet switches must maintain full routing tables and be capable of making route computations on the fly at wire speed. In addition wider market pressures have often led to switches including extra features such as layer 3 and

higher based routing that are unneeded in the cluster world.

Gigabit Ethernet has been used as a high speed interconnect for a number of years. However, early on its performance in PC systems was limited to 300-500 Mbps. In addition Gigabit Ethernet switches were expensive and of limited port count. With the arrival of copper based NICs and inexpensive switches it became the low-cost solution for clusters up to 24 nodes. Beyond that size switches remained very expensive until very recently. Today, driven partly by the arrival of 10 Gigabit Ethernet, high-density Gigabit Ethernet switches are an affordable solution through around 480 nodes.

One significant issue with Ethernet has always been the relatively high CPU overhead of a full TCP/IP stack. This issue greatly limited performance on early systems. One common, but non-standard technique is to increase the Maximum Transmittable Unit (MTU). Standard Ethernet has always specified an MTU of 1500 bytes regardless of the speed. This creates a large overhead associated with packetization that can greatly impact performance. It has become fairly common to increase the MTU to 9000 bytes (aka Jumbo Frames), which has the effect of reduces the packetization overhead by a factor of six. However, all of the NICs and switches in the system must support the larger MTU size. Today CPUs and system busses have progressed to the point where most systems can achieve 90% or better utilization even using the standard MTU.

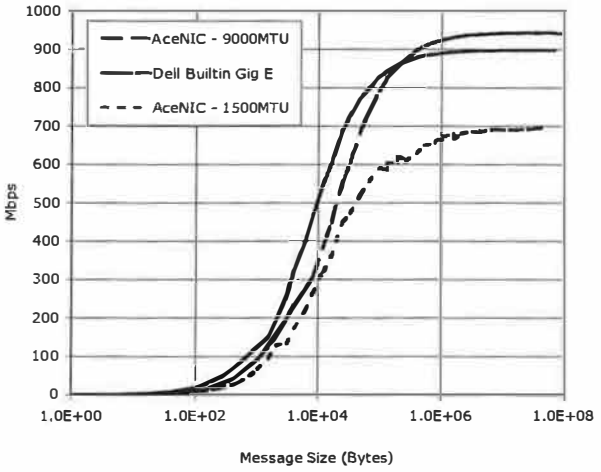



**Figure 1**: Gigabit Ethernet Performance

Figure 1 illustrates the performance of two Gigabit Ethernet NICs. The first and older NIC is based on the Alteon ACENIC chipset. The ACENIC provides a flexible CPU based architecture capable of jumbo frame support. Unfortunately the CPU based architecture has relatively high latency at around 90 $\mu$seconds. Partly due to the high latency the Alteon NIC requires the use of Jumbo Frames to achieve good results, but went Jumbo Frames are used performance around 940Mbps is seen. The second NIC is a built-in NIC on the Dell 2650 motherboard based on a Broadcom ASIC based chipset. The Broadcom chipset has become quite common and due to its low cost is the basis for many low cost as well as integrated Gigabit Ethernet solutions. As Figure 1 shows the Broadcom part performs quite well with a latency of 31 $\mu$seconds, which is excellent for any Ethernet NIC. In addition the lower latency translates into significantly better performance in the 10KB region. Finally the peak performance of 900 Mbps is quite good especially since it was achieved with the standard 1500 byte MTU.

A second issue is the inability to aggregate multiple switches in such a way as to provide full bi-section bandwidth. While most switches do provide the ability to trunk multiple ports together between switches this is both expensive and generally inadequate effectively limiting the size of an Ethernet based cluster to the size of the largest available switch, which is currently around 480 ports.

The Cost of Gigabit Ethernet has come down substantially over the last few years. While the Alteon, etc CPU based NICs are still relatively pricey at $300/NIC, the ASIC based NICs such as the Broadcom based products are widely available at less than $100/NIC, if they aren't integrated onto the motherboard. For small clusters there have been inexpensive (<$100/port) switches available for some time. However the price of the larger switches has now dropped substantially as well. For moderate size clusters (64-128 ports) per port pricing runs $200-$300/port while high-end switches run around $667/port. This puts the total cost of a Gigabit Ethernet solution at $150/port at the low end up to around $750/port at the high end which certainly makes it the lowest cost option considered in this paper.

## 2.2. Myrinet[2]

Myrinet was one of the first interconnect technologies designed specifically with clustering in mind. Because of this design criterion it makes some tradeoffs not possible in more general-purpose technologies such as Ethernet. The main feature of the design is that packets are source routed over a fat-tree based network made up of relatively small switch elements (16 port switches). This obviously requires that each node know the full network topology or map and that it be fairly static.

The big payoff is that the switch elements can be very simple since they do not perform any routing calculations. In addition the software design is based on an OS-bypass like interface to provide low-latency and low-CPU overhead.

Current Myrinet hardware utilizes a 2 Gbps link speed with PCI-X based NICs providing one or two optical links. The dual-port NIC virtualizes the two ports to provide an effective 4 Gbps channel. The downside to the two port NIC is cost. Both the cost of the NIC and the extra switch port it requires. Myrinet is designed to be scalable. Until recently the limit has been 1024 nodes, but that limit has been removed in the latest software. However, there are reports that the network mapping is not yet working reliably with more than 1024 nodes. The cost of Myrinet runs around $850/node up to 128 nodes, beyond that a second layer of switches must be added increasing the cost to $1737/node for 1024 nodes. The dual port NICs effectively double the infrastructure and thus add at least $800 per node to the cost.

Myricom provides a full open-source based software stack with support for a variety of OS's and architectures. Though some architectures, such as PPC64, are not tuned as well as others. One of the significant plusses for Myrinet is its strong support for TCP/IP. Generally TCP/IP performance has nearly matched the MPI performance, albeit with higher latency.

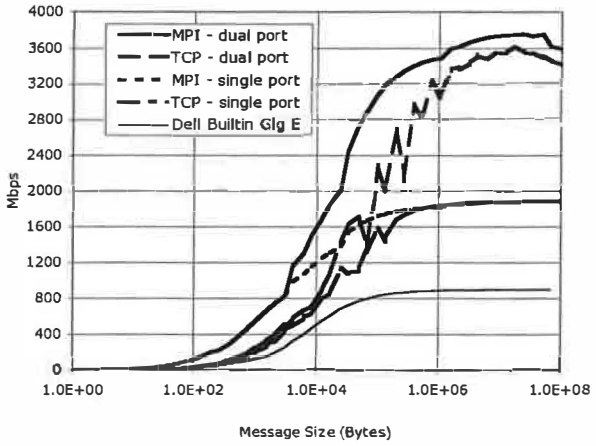



**Figure 2: Myrinet Performance**

Figure 2 illustrates the performance of both the single and dual port NICs with the Broadcom based Gigabit Ethernet data included from Figure 1 for reference. In both cases the MPI latencies are very good at 6-7 $\mu$seconds and TCP latencies of 27-30 $\mu$seconds. Also both NICs achieve around 90% of their link bandwidth with over 3750 Mbps on the dual NIC and 1880 Mbps on the single port NIC. TCP/IP performance is also excellent with peak performance of 1880 Mbps on the single port NIC and over 3500Mbps on the dual port NIC.

## 2.3. Scalable Coherent Interface[3]

The Scalable Coherent Interface (SCI) from Dolphin solutions is the most unique interconnect in this study in that it is the only interconnect that is not switched. Instead the nodes are connected in either a 2D wrapped mesh or a 3D torus depending on the total size of the cluster. The NIC includes intelligent ASICs that handle all aspects of pass through routing, thus pass through is very efficient and does not impact the host CPU at all. However, one downside is that when a node goes down (powered off) its links must be routed around thus impacting messaging in the remaining system.

Since the links between nodes are effectively shared the system size is limited by how many nodes you can effectively put in a loop before it is saturated. Currently that number is in the range of 8-10 leading to total scalability in the range of 64-100 nodes for a 2D configuration and 640-1000 nodes for the 3D torus. Because there are no switches the cost of the systems scales linearly with the number of nodes, $1095 for the 2D NIC and $1595 for the 3D NIC including cables. Unfortunately cable length is a significant limitation with a preferred length of 1m or less, though 3-5m cables can be used if necessary. This poses quite a challenge in cabling up a systems since each node must connected to 4 or 6 other nodes.

Dolphin provides an open source driver and a 3rd party MPICH based MPI is under development. However,

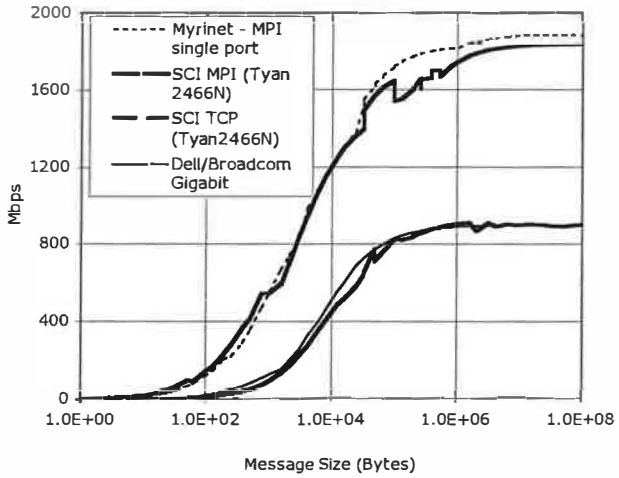



**Figure 3: SCI Performance**

currently we have not gotten the MPI to function correctly in some cases. An alternative to the open source stack is a package from Scali[4]. This adds $70 per node, but does provide good performance. Unfortunately the Scali packages are quite tied to the Red Hat and Suse distributions that they support. Indeed it is difficult to get the included driver to work on kernels other than the default distribution kernels.

Figure 3 illustrates the performance of SCI on a Tyan 2466N (dual AMD Athlon) based node. Since Dolphin does not currently offer a PCI-X based NIC it is unlikely that the performance would be substantially different in the Dell 2650 nodes used for the other tests. The Scali MPI and driver were used for these tests. The MPI performance is quite good with a latency of 4 $\mu$seconds and peak performance of 1830Mbps nearly matching that of the single port Myrinet NIC, which is a PCI-X, based adapter. However, the TCP/IP performance is much less impressive as it barely gets above 900 Mbps and more importantly proved unreliable in our tests.

## 2.4. Quadrics[5]

The Quadrics QSNET network has been known mostly as a premium interconnect choice on high-end systems such as the Compaq AlphaServer SC. On systems such as the SC the nodes tend to be larger SMPs with a higher per node cost than the typical cluster. Thus the premium cost of Quadrics has posed less of a problem. Indeed some systems are configured with dual Quadrics NICs to further increase performance and to get around the performance limitation of a single PCI slot.

The QSNET system basically consists of intelligent NICs with an on-board IO processor connected via copper cables (up to 10m) to 8 port switch chips arranged in a fat tree. Quadrics has recently released an updated version of their interconnect called QSNet II based on ELAN4 ASICs. Along with the new NICs Quadrics has introduced new smaller switches, which has brought down the entry point substantially. In addition the limit on the total port count has been increased to 4096 from the original 1024. Still it remains a premium option with per port costs starting at $2400 for a small system, $2800 per port for a 64 way system, up to $4078 for a 1024 node system.

On the software side quadrics provides an open source software stack including the driver, userland and MPI. The DMA engine offloads most of the communications onto the IO processor on the NIC. This includes the ability to perform DMA on paged virtual memory addresses eliminating the need to register and pin memory regions. Unfortunately their supported software configuration also requires a licensed, non-open source resource manager (RMS). In our experience the RMS system was by far the hardest part to get working.

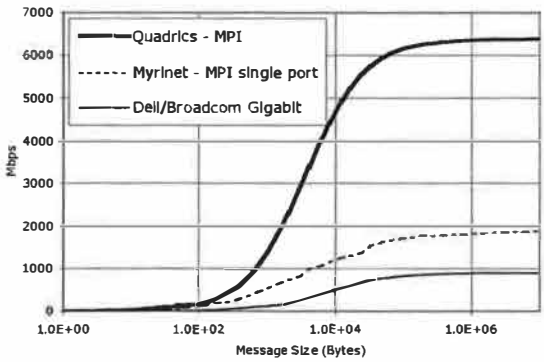



**Figure 4: Quadrics Performance**

Figure 4 illustrates the performance of Quadrics using the MPI interface. TCP/IP is also provided, but we were unable to get it to build on our system in time for this paper due to incompatibilities with our compiler version. The MPI performance is extremely good with latencies of 2-3 $\mu$seconds and peak performance of 6370 Mbps. Indeed this is the lowest latency we have seen on these nodes.

## 2.5. Infiniband

Infiniband has received a great deal of attention over the past few years even though actual products are just beginning to appear. Infiniband was designed by an industry consortium to provide high bandwidth communications for a wide range of purposes. These purposes range from a low-level system bus to a general purpose interconnect to be used for storage as well as inter-node communications. This range of purposes leads to the hope that Infiniband will enjoy commodity-like pricing due to its use by many segments of the computer market. Another promising development is the use of Infiniband as the core system bus. Such a system could provide external Infiniband ports that would connect directly to the core system bridge chip bypassing the PCI bus altogether. This would not only lower the cost, but also provide a significantly lower latency. Another significant advantage for Infiniband is that is designed with scalable performance. The basic Infiniband link speed is 2.5Gbps (known as a 1X link). However the links are designed to bonded into higher bandwidth connections with 4 link channels (aka a 4X links) providing 10Gbps and 12 link channels (aka 12X links) providing 30 Gbps.

Current Infiniband implementations are available as PCI-X based NICs and 8 or 24 port switch chips utilizing 4X (10Gbps) links. The switch chips can be configured for other link speeds as well including 12X links. This makes switch aggregation somewhat easier since you can configure a switch with 12 4X links to connect to nodes and 4 12X links to connect to other switches. Current Infiniband pricing is enjoying the benefits of aggressive venture capital funding while the various vendors attempt to define their market. Thus there are a variety of vendors competing with slightly different NICs and switches even though the core silicon in most current implementations is from Mellanox[6]. Current pricing ranges from $1200-$1600/per depending on the vendor and cluster size (pricing would be higher for very large clusters).

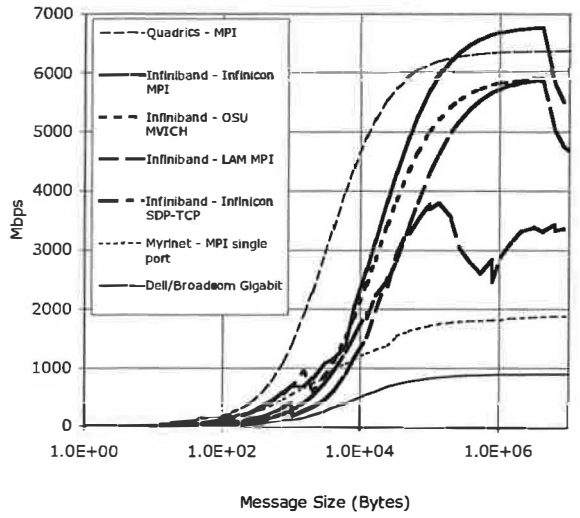



**Figure 5: Infiniband Performance**

One of the primary ways the vendors are attempting to differentiate their products is through their software stack. This has created a general reluctance to release the software stack as full open-source. In addition many of the software elements are still very much in development. For our tests we attempted to us a couple of MPI implementations, but found the latest MPICH2 based code unstable. Instead we used an older MVICH based implementation. Also included is a non-open source stack from Infinicon Corporation[7].

Figure 5 plots the performance of Infiniband versus a couple of other technologies. Clearly there are substantial differences between the different MPI implementations. The Infinicon MPI shows quite good peak performance over 6750 Mbps, the other two peak out around 5900 Mbps. All three show a large drop off above 4MB in message size when the message size exceeds the on NIC cache size. In addition current latencies of 6-7 $\mu$seconds are a bit higher than the other OS-bypass technologies.

## 2.6. 10 Gigabit Ethernet

10 Gigabit Ethernet is the next step in the Ethernet series. Because of its heritage its design inherits all the advantages and disadvantages of previous Ethernet implementations. The primary advantages include interoperability with previous Ethernet generations, wide portability and a ubiquitous interface (TCP/IP). The primary disadvantages have always been relatively high latency and CPU load, as well as expensive switches. In the present case a portion of the cost is being driven by the cost of the optics which currently run around $3000 by themselves. Some Gigabit Ethernet switches are now offering 10 Gigabit uplink ports essentially at the cost of installing the optics that is making it somewhat more practical to trunk multiple Gigabit Ethernet switches together. However, full 10 Gigabit Ethernet switches are still fairly expensive at around $10000 per switch port even though that figure has dropped by a factor of 3-5 in the last year. In addition 10 Gigabit Ethernet switches are currently fairly limited in port count with large switches offering only around 48 ports. The cost and low-port density clearly make cluster based on 10 Gigabit Ethernet unlikely in the near term. In the longer term these issues will likely mitigate as the technology commoditizes.

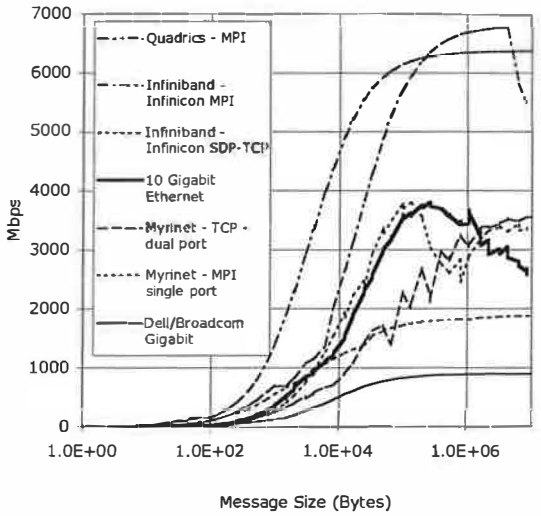



**Figure 6: 10 Gigabit Ethernet Performance**

Figure 6 plots the performance of two Intel 10Gigabit Ethernet ports against samples of the other technologies. Clearly the performance of 10 Gig. E. is disappointing with performance topping out around

3700Mbps. However this does appear a common limit of TCP/IP among the other technologies and thus may be in part limited by the overhead of TCP/IP itself.

## 3. Application Performance

Beyond raw network performance there are other issues that can dramatically effect the overall performance of the interconnect. Chief among the issues is how the software and hardware deal with running more processes than there are processors. Especially in the case of applications that use auxiliary processes as data servers. This type of processes typically must block in a receive call. Under many MPI implementations this results in a polling behavior that can take substantial CPU time away from the compute processes. In our local environment the dominant application is the quantum chemistry package GAMESS[8]. Thus we have ran a sample, communication intensive, calculation on each of the interconnect choices to see just how much impact the network has on the timings.

| #node, 2Procs per | 1 | 2 | 4 |
|---|---|---|---|
| Gigabit Ethernet | 5026 | 1961 | 1529 |
| Myrinet MPI | 3277 | 1984 | 1559 |
| Myrinet TCP | 3477 | 1924 | 1279 |
| Infiniband MPI | 4868 | 2785 | 1319 |
| Quadrics MPI | | 2593 | 1427 |
| **# node, 1 proc per** | **1** | **2** | **4** |
| Gigabit Ethernet | 5823 | 3062 | 1852 |
| Myrinet MPI | 5651 | 3030 | 1743 |
| Myrinet TCP | 5843 | 3059 | 1801 |
| Infiniband MPI | 5743 | 3006 | 1727 |
| Quadrics MPI | | 3052 | 1775 |

**Table 1: GAMESS results**

Table 1 lists some timings for a calculation that utilizes a dual process model. One process functions as the compute process and the second process acts as a data server for a pseudo global shared memory segment. The first block of timings overload the CPUs with 2 compute and 2 data servers on each dual CPU node. The second block run only 1 compute and 1 data server per node.

One obvious result is that the fastest network does not necessarily produce the fastest timing for this application. When the CPU's are not overloaded (ie one compute and one data server per node) the timings are fairly similar for all interconnect types. However when the number of processes per node is doubled the results are more interesting. First off Myrinet turns in substantially faster timing for a single node. Most likely this indicates a good intra-node message passing implementation. However when running on 4 nodes the Myrinet TCP run is faster than the others. The reason the MPI's are likely slower is due to the way many MPI's handle blocking in a receive call. Many implementations assume that each MPI process essentially owns a CPU and thus implement a polling mechanism that eats CPU time. For applications such as GAMESS this results in a substantial loss of performance. In addition many of the interconnects seem to perform worse when there are lots of processes utilizing the interconnect at the same time. By this we mean there seems to be substantial overhead in multiplexing and demultiplexing the messages when multiple processes are simultaneously active.

## 4. Conclusions

The good news is that today there are several good choices for a high-speed interconnect on a cluster at a range of price. At the low-end Gigabit Ethernet has emerged as a solid option with a cost of $750/node or below up to several hundred nodes. Of course Gigabit Ethernet is also the lowest performing network in the survey, but its solid, ubiquitous, software support make it a good choice for applications that do not require cutting edge communications performance.

In the middle of the pack in terms of cost and performance are SCI and Myrinet. Both products offer good performance at a moderate cost. SCI has some interest due to its flat cost per node as the cluster size is scaled up. However, the software stacks available for SCI have several issues, some of which significantly impact the performance of the system. Myrinet on the other hand offers, a complete open source software package that is the most solid software stack surveyed apart from TCP/IP based Ethernet. The current Myrinet hardware provides good performance at a reasonable cost and is thus a quite solid choice for most applications.

Infiniband stands out as a technology with a great deal of promise, but quite a few rough edges. Chief among the rough edges are the problems with the varied software stacks. Currently it is possible to setup an Infiniband based network and find a usable software stack. However, such a network will require a bit more maintenance as the software is revised.

In the long term 10 Gigabit Ethernet will likely play a role in clusters, but it will likely take at least a couple more years before the price becomes competitive. During that time CPUs and busses will also increase in

also be likely to deliver a more acceptable percentage of its theoretical bandwidth.

At the very high-end of the spectrum, in terms of both performance and cost, lies the Quadrics interconnect. It provides very low latency and very good peak performance, but its cost comes in at nearly twice its competitors. This will likely continue to leave Quadrics as a niche product used only on very high-end systems with either big SMP nodes (ex 4 way Alpha or IA64 systems) or those requiring the ultimate in scalability. Even in these cases it would be nice to see the dependence on the licensed RMS software removed.

## 5. Acknowledgements

This work was performed under the auspices of the U.S. Department of Energy under contract W-7405-Eng-82 at Ames Laboratory operated by the Iowa State University of Science and Technology. Funding was provided by the Mathematical, Information and Computational Science division of the Office of Advanced Scientific Computing Research.

---

[1] Quinn, Snell O.; Mikler, Armin R.; Gustafson, John L; Helmer, Guy. "NetPIPE: a Network Protocol Independent Performance Evaluator". Scalable Computing Laboratory, Ames Laboratory. http://www.scl.ameslab.gov/Projects/NetPIPE/index.html.

[2] http://www.myricom.com/

[3] http://www.dolphinics.com/

[4] http://www.scali.com/

[5] http://www.quadrics.com/

[6] http://www.mellanox.com/

[7] http://www.infinicon.com/

[8] M.W.Schmidt, K.K.Baldridge, J.A.Boatz, S.T.Elbert, M.S.Gordon, J.H.Jensen, S.Koseki, N.Matsunaga, K.A.Nguyen, S.J.Su, T.L.Windus, M.Dupuis, J.A.Montgomery *J. Comput. Chem.* **14**, 1347-1363 (1993).

# InfiniBand Performance Review: It's the Software Stupid

*Troy R. Benjegerdes, Brett M. Bode*
Scalable Computing Laboratory
Ames Laboratory
*troy@scl.ameslab.gov, brett@scl.ameslab.gov*
*http://www.scl.ameslab.gov/*

## Abstract

The InfiniBand interconnect has received a great deal of attention recently as a result of its use in the Virginia Tech cluster that placed 3rd on the Top500 list. However, InfiniBand hardware and software have been available for over a year and continue to evolve. In addition, InfiniBand is supported by a number of different vendors each of which is seeking to differentiate themselves in the marketplace with slightly different hardware and software offerings. This paper will examine many of the current hardware and software implementations, illustrating their similarities and differences. In addition we will demonstrate the sometimes dramatic effects of the various tuning parameters of the various software implementations.

## 1 Introduction

The InfiniBand interconnect has seen a great deal of publicity in the past few years since its inception. Once the initial marketing faded many thought the whole concept had failed. In fact a great deal of effort was continuing to make the concept a reality. For a little over a year, 4X InfiniBand (10Gbps) hardware has been available from a variety of vendors. During that time the software stacks have matured a great deal to the point where it is now practical to use InfiniBand as the primary interconnect in a production oriented High-Performance Computing (HPC) system. Indeed the number 3 system on the top500 [ref top500.org] list is now an 1100 node cluster connected by InfiniBand.

This is not to say there is nothing left to be done. On the contrary, one of the biggest problems for the InfiniBand community is the software available. Up until early spring in 2004, each InfiniBand vendor was providing their own proprietary software stack. This software required a specific kernel binary from a distribution like Red Hat or Suse, and were generally only available x86 platforms. In our case, we have x86, amd64, and ppc64 platforms, so this was far from optimal. Recently, several vendors have released their hardware drivers and software stacks as open source. While this is a good step, it is much like the first releases of the Netscape source code as open source. Yes, it's out there and available, but it's not something anyone other than a dedicated hacker is really going to use.

Part of this review will cover the impressive performance results obtained in November of 2003, using vendor provided software stacks and MPI implementations. We will also examine performance of the latest low level InfiniBand driver stack from Mellanox, which is expected to be released soon under a dual GPL/BSD license. In addition, we will briefly examine application behavior of the GAMESS computational chemistry application on a 4 node InfiniBand cluster.

Finally, we will discuss some of problems with the currently available open-source InfiniBand stacks. Some of these problems include difficulty in the build process on systems not explicitly supported by the vendor, issues with multiple architecture support, and the disconnect between the larger network research community and the InfiniBand community.

## 2 History

First, some history of our experience with InfiniBand. Our first real exposure was at the InfiniBand Birds of a Feather at Ottawa Linux Symposium 2001 [InfiniBand BOF]. Af-

ter this, we obtained InfiniBand HCA's and a switch development platform from Mellanox [Mellanox Technolgies].

At one point, both IBM and Intel had plans for making Host Channel adapters. Unfortunately, due to the 'dot-bomb' phenomenon, and other reasons known only to IBM and Intel decision makers, both companies canceled their plans for InfiniBand host adapters. However, Intel did keep on a team of software engineers on staff and open sourced their access layer. This code base was placed into a sourceforge project, and is generally referred to as the Intel Verbs layer, or IBAL [IBAL]. Unfortunately, since there was no shipping hardware, this access layer project got little attention outside of Intel.

## 2.1 InfiniBand finally delivers

Initially, Mellanox's low level drivers and firmware were still in early development releases, and performance was not that impressive. However, right before SuperComputing 2002, a succession of new firmware and drivers pushed the peak bandwidth achievable up to over 6 gigabits. This peak bandwidth was far above performance achievable on Myrinet, SCI/Dolphin, or 10 Gigabit Ethernet at the time. At this time, Dr. D.K. Panda's group at Ohio State University released their implementation of MPICH [MVAPICH] for Mellanox's VAPI InfiniBand software stack.

By SuperComputing 2003, several vendors, including InfiniCon had integrated together a software stack, based on the Mellanox low level drivers, an InfiniBand access layer, and MPI implementations based on MVIAPICH [InfiniCon]. There was also a large demand, particularly among the DOE laboratories and third party vendors, like Oracle, for Open Source InfiniBand drivers. At this point, all the major vendors were shipping products based on the Mellanox InfiniBand Host Channel Adapter (HCA). This led to vendors being reluctant to support an open-source solution, since they were attempting to differentiate and add value in the software stack.

## 2.2 Where's the source

The reluctance by vendors to open source was compounded by the fact that they were all using the same low level drivers provided by Mellanox, so even if they were to open source their access layer, in practice it would be useless without a low level driver. The IBAL sourceforge project currently still does not have a working open-source low level driver.

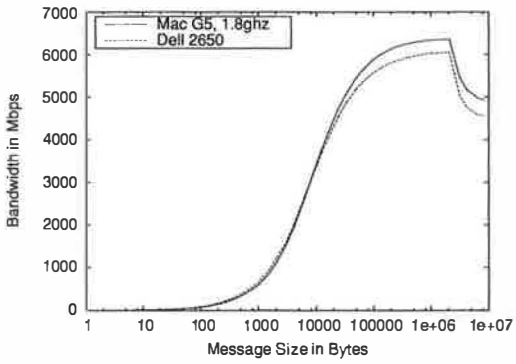



Figure 1: Raw Mellanox VAPI performance

Finally, in April of 2004, several InfiniBand vendors, including Topspin, InfiniCon, Voltaire, Mellanox, and Divergenet all announced, and released portions of their respective software stacks and drivers under various open-source licenses.The openib.org website was set up by a collaboration between InfiniBand vendors, and various US DOE Labs.

In some ways, the current situation resembles that of the initial Netscape source code release. There is a lot of code available for download from various places now, but none of it is something that can be used in a production cluster or data center environment. It is, however, a good opportunity for research.

# 3 Performance

Raw performance delivered by current Mellanox-based InfiniBand hardware is quite impressive and peak bandwidth is primarily limited by the PCI-X bus implementation on the system. Figure 1 illustrates the raw performance available using the Mellanox VAPI interface on a 2.4ghz Dell 2650 and 1.8ghz Macintosh G5.

## 3.1 Hardware test environment

For most of the data presented, the hardware test environment consisted of a cluster of 4 Dell 2650 dual Xeon systems, two Macintosh G5's, and two dual AMD Opteron systems. Mellanox-based HCA's from three vendors have been tested, and no noticeable performance difference has been noticed between vendors. All the software stacks tested have also worked on any HCA card with a Mellanox ASIC.

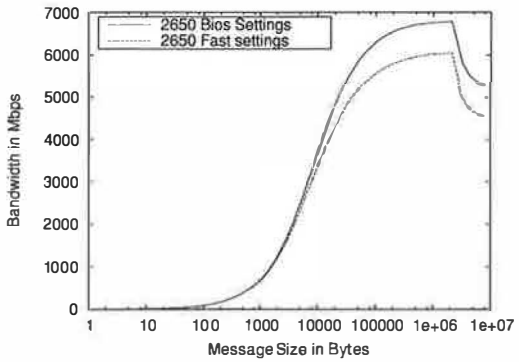



Figure 2: Serverworks chipset flag impact

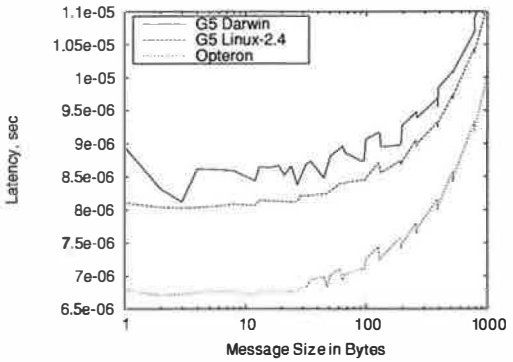



Figure 3: Raw VAPI latency with NetPIPE

The Dell 2650 systems consist of 2 2.2Ghz dual Xeon systems, and two 2.4Ghz dual Xeon systems purchased a year later, which have the faster 533mhz front-side bus. Most tests have been run on the 2.4Ghz machines. These systems have the ServerWorks Grand Champion chipset. These systems have a chipset flag which can be set by the Linux 'setpci' program which significantly increases peak PCI-X bandwidth, however it is reported setting this flag results in stability problems. Figure 2 shows the performance impact of setting, for lack of a better term, the Serverworks benchmark bit.

The Macintosh G5 test system consisted of a 1.8Ghz Mac G5, and a dual 2.0Ghz Mac G5. Performance is very similar to that of the AMD Opteron systems, which consisted of two Dual 1.4Ghz AMD Opterons with 8GB of memory each. This is expected since both systems use the AMD 8131 HyperTransport to PCI-X bridge chip. The only appreciable difference is the latency, which is to be expected since the G5 system has a bridge chip (the Uni-N ASIC) between the HyperTransport (HT) and CPU's, while the Opteron system has native HT links on the CPU. Figure 3 shows the difference in latency between the Opteron and G5 systems, taken from the November 2003 data. It is unclear why OSX(Darwin) has a higher latency than Linux on the same hardware.

## 3.2 Software test environment

Our first priority for our most recent round of InfiniBand performance tests was to reduce the overhead of installation and system maintenance. Due to this, we used the debian linux based NFS-root file system images we have already set up for other clusters. This requires that any InfiniBand drivers be buildable from source code, and run on the latest linux kernel (at this point linux-2.6.5, or linux 2.4.26), and work on the Debian linux distribution running from an NFS-root mounted filesystem. This quickly eliminated a number of the vendor provided solutions. The linux-2.4.26 kernel used for testing on the dell 2650's was also patched with Quadrics QSNet kernel patches due to testing Quadrics on the same machines.

Due to time constraints, we wound up only being able to run the pre-release thca-3.2-rc9, since this was the source base we had the most experience with trying to tweak it to run on our three types of test systems. In addition, with the exception of the Divergenet stack, all the other vendors are based on some Mellanox thca release, so this was the natural place to start. Due to the recent level of activity on openib.org, and internal vendor projects we expect we may have new results on other stacks by the time this paper is presented.

The AMD Opteron systems were running a debian-amd64 biarch system, with a 32 bit base system, and specific 64 bit libraries. All the InfiniBand libraries were built as 64 bit libraries since the Mellanox thca release does not have any facilities for biarch 32/64 bit environments, and the kernel code is 64 bit. Due to a build problem, we were unable to build a 2.6.5 kernel for Opteron at this time.

We were only able to obtain results on the Macintosh G5 on a 32 bit linux-2.4 kernel and on MacOSX, which is also 32 bit. After some changes, it is possible to get the Mellanox thca to build for a PPC64 linux environment, however the module does not load due to attempting to access a very low level memory management primitive. It is unclear if this is a generic InfiniBand issue, or something specific to PPC64. Even if

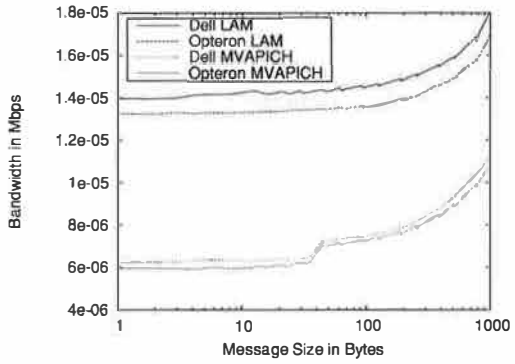



Figure 4: MPI Latency

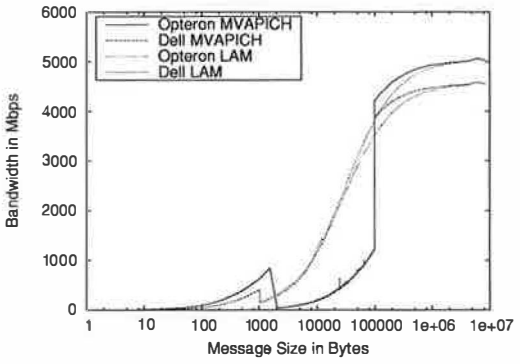



Figure 5: MVAPICH bad interaction with caches

the module were to load, it, it's not clear it would work due to issues with the PCI-X bridge and having to use an IOMMU. These problems are a rather strong indication that the current InfiniBand software was written primarily with x86 Intel systems and time-to-market considerations in mind rather than cross-platform software portability.

## 3.3 MPI testing

For MPI testing, we tested both the OSU-0.9.2 MVAPICH patches to MPICH-1.2.5, and an April 28 checkout of the LAM-MPI subversion repository. Figure 4 shows that MVAPICH, which uses an RDMA-write and memory polling, has significantly lower latency than LAM. MVAPICH also has a lower latency that NetPIPE-3.6's raw IB module, due to NetPIPE using POLL_CQ, which polls the InfiniBand card, causing extra PCI-X cycles.

The MVAPICH patches to MPICH are more mature, and have been available for over a year. They have been derived from the earlier M-VIA work at Berkeley Lab [M-VIA], and have several optimizations that LAM-MPI lacks. However, as Figure 5 shows, there are cases where maturity and advanced optimizations lose out to a simpler implementation.

These dropouts in MVAPICH only occur when running NetPIPE-3.6 with the '-I' cache-invalidate option, which causes NetPIPE to rotate through many buffers when sending ping-pong messages instead of re-using the same buffer. This forces worst-case behavior by causing a cache miss on every subsystem in the message path. In this case, there are internal caches in the MPI implementation for re-using so-called "eager" buffers, as well as a translation protection

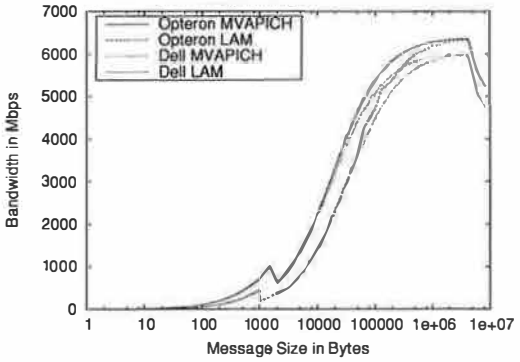



Figure 6: MVAPICH and LAM-MPI performance

table (TPT) cache in the Mellanox ASIC. The TPT cache configuration is also the reason for the dropouts from around 6 gigabits to 5 gigabits on messages larger than 1-2MB on other graphs as well. Figure 6 shows the results of a NetPIPE run in which the messages are sent from the same buffers every time. MVAPICH makes effective use of internal caches and the TPT cache on the HCA as well, showing noticeable better performance than LAM-MPI at medium message sizes.

The dropouts in MVAPICH are not, however, inherent to the code base. Figure 7 shows the performance of the MVAPICH and InfiniCon MPI from our earlier November 2003 data. The code base is largely the same, since InfiniCon used MVAPICH as a base. The differences appear to result from differences in tuning parameters, interaction with the memory management subsystem, and the Mellanox TPT cache. With the cache-invalidate option, InfiniCon's MPI only has two small dropouts around 10K byte messages sizes, opposed to MVAPICH, which has a drop from around 2K to 100k. Differences in

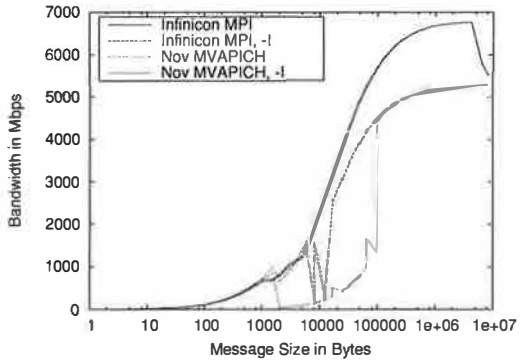



Figure 7: MVAPICH and InfiniCon MPI

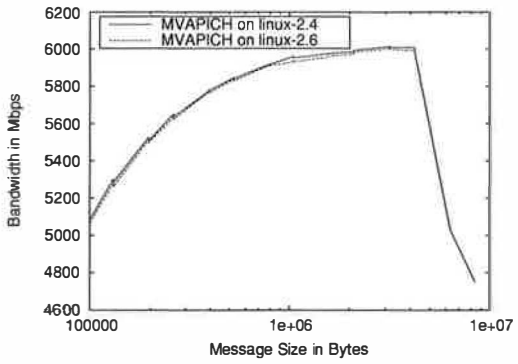



Figure 9: Kernel 2.4 vs 2.6, big messages

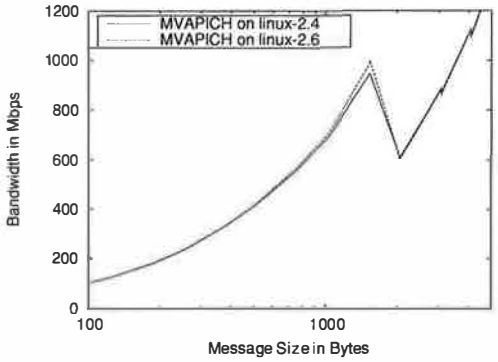



Figure 8: Kernel 2.4 vs 2.6, small messages

peak bandwidth from other graphs are due to the Serverworks chipset options discussed earlier.

### 3.4 Kernel versions

Figure 8 and Figure 9 show very little difference between a linux-2.4 series kernel, and linux-2.6.5 running on the Dell 2650 hardware. This is to be expected since the current driver implementations bypass many of the linux kernel subsystems.

## 4 Conclusions

We have seen that InfiniBand can achieve very good performance. What remains is to see how well it continues to evolve and compete with other custom cluster interconnects. Vendor support for small clusters and 'commercial' linux distributions like Red Hat and Suse seems to be very good. What's missing is better support for open source, and a common API for application development.

### 4.1 A Commodity Interconnect?

Many claims have been made over the history of InfiniBand about it becoming the 'new' commodity network interconnect. The first several years of it's existence were met with a great deal of skepticism from various communities that InfiniBand was just another marketing buzzword, and a bunch of vaporware. Now, the hardware exists, and proprietary driver stacks seem to work well in specific environments. Current pricing on HCA's is around $750 a port, and switch pricing is $300 a port. While this pricing is quite competitive for other cluster-specific interconnects, it is dependant on InfiniBand adoption in the Data Center and Enterprise Storage markets. It remains to be seen whether the potential of commodity volume production and raw performance is offset by the additional complexity of the software stack. 10 Gigabit Ethernet has a definite advantage in that drivers for the Intel 10Gig-E card have been in the linux-2.6 kernel series for several months, and other vendors have submitted drivers for inclusion into the mainline kernel.

### 4.2 Linux integration

InfiniBand's biggest (and some would say fatal) flaw is the amount of new code required to just get something to run. The Mellanox low level driver alone is over 100,000 lines of code. This doesn't count extras like sockets direct, SCSI Remote Protocol, or an IP over IB driver. We have some very nice OS-bypass hardware, but it requires what amounts to half an OS worth of additional software to run. The hoped-for commodity markets of scale will never occur unless adding InfiniBand drivers is not much different than adding an ethernet driver. In the case

of Linux, this means integration into the memory management subsystem in a clean, cross-platform manner, and a minimal driver that can bring the card up and send packets without a lot management code.

There is a definite opportunity here to develop a clean API for RDMA-capable networks like InfiniBand, and get that API integrated into Linux. But it's got to be something for more than just InfiniBand. 10 Gigabit Ethernet is going to need some sort of RDMA capability to function well, and the existing high-performance cluster networks could benefit from something like this as well. The real benefit isn't necessarily to the network vendor, it's to application developers who currently use the Berkeley Sockets API, because it's the only thing that's portable. Sockets direct is appealing, but in order for it to work, there needs to be a consensus on how it is to work across different types of networks.

## 5 Acknowledgments

This work was performed under auspices of the U. S. Department of Energy under contract W-7405-Eng-82 at Ames Laboratory operated by the Iowa State University of Science and Technology. Funding was provided by the Mathematical, Information and Computational Science division of the Office of Advanced Scientific Computing Research.

We would also like to thank InfiniCon Systems for a hardware loan, Jeff Kirk at Mellanox for invaluable technical assistance, everyone contributing to the OpenIB.org project, and linux kernel developers who have taken an interest in InfiniBand.

## References

[InfiniBand BOF] InfiniBand on Linux BOF, *http://www.linuxsymposium.org/2001/bofs.php* Ottawa Linux Symposium, 2001

[Mellanox Technolgies] *http://www.mellanox.com*

[IBAL] *http://infiniband.sourceforge.net*

[MVAPICH] *http://nowlab.cis.ohio-state.edu/projects/mpi-iba*

[InfiniCon] *http://www.infinicon.com*

[M-VIA] *http://www.nersc.gov/research/FTG/via/download_info.html*

# A new Distributed Security Model for Linux Clusters

Makan Pourzandi
*Open Systems Lab, Ericsson Research, Town of Mount-Royal (QC) Canada.*
Makan.Pourzandi@Ericsson.Com

**Abstract**

With the increasing use of clusters in different domains, efficient and flexible security has now become an essential requirement for clusters, though many security mechanisms exist, there is a need to develop more flexible and coherent security mechanisms for large distributed applications.

In this paper, we present the need for a unified cluster wide security space for large distributed applications. Based on these needs, we propose a new security model that implements security zones inside the cluster. The model is an extension to Mandatory Access Control (MAC) mechanisms used at node level to the whole cluster with processes as basic security entities.

We designed this model with clustered Linux servers running carrier-grade applications in mind but this model can be used in any domain that needs Linux clusters running large distributed applications continuously with no interruptions. We prove the feasibility of this approach through an open source implementation of the concept [1].

## 1 Introduction

Distributed applications become more and more complex. Often, they are made of different software components of different functionality. The current security mechanisms are based on user permissions, that provides solid security environment in many cases but they are not flexible enough for large distributed applications.

In many of these applications, only few users exist on dedicated clusters for running a pre-defined set of software for a long period of time without interruption. However, the user based security system does not support authentication and authorization checks for interactions between two processes belonging to the same user. This situation leads to an all-or-nothing approach, as all users within a group or all processes of the same user have the same rights. This is most inconvenient when one wishes to compartmentalize these rather large distributed applications by restricting the access to some resources to some processes or users of the same group.

This lack of compartmentalization between different software components, results in that a vulnerable small component may compromise the whole system. This situation gets even worse with systems running untrusted software (unfortunately, in practice, it is impossible for time and economic reasons to security audit all software running for these rather large applications).

This situation is due to the use of user-level granularity as the basic entity for the security control in these distributed applications, for which this granularity is not sufficient. Therefore, there is a need for finer granularity security mechanisms which use the individual process as the basic entity.

Furthermore, the security must be pervasive and make a coherent system across the cluster. Therefore, distributed security functions must be put in place throughout the cluster. However, the current situation is based on assembling heterogeneous security solutions for different nodes. This leads to security management nightmares and stiff integration problems, and too often leaves gaps between different security mechanisms allowing security breaches.

The paper is organized as follows: Section 2 discusses the main concepts behind our approach. In Section 3, we detail the implementation of our security model. We conclude by presenting the future work.

## 2 Cluster-wide security space

We detail hereafter different elements of our security model.

### 2.1 Process level granularity

In order to implement process-level security mechanisms, we need to identify the different security contexts for individual processes inside the cluster.

A *security node identifier* (SnID) is assigned to each node. All processes and resources also receive a *security context identifier* (ScID). We define ScIDs for processes, binary files and resources on the cluster. These secure IDs are coherent and meaningful within the entire cluster. Also, ScIDs are persistent (they do not change after rebooting the system). Actually, one should think of ScIDs more like security GIDs than PIDs: ScIDs are meant to group together processes and resources that have the same security context.

Any newly created process is assigned a ScID which is based on the ScID of the parent process, the ScID stored in the loaded binary, and the general security context of the system.

In our security model, all different security mechanisms (access control, authentication, confidentiality, integrity and logging) are based on the pair (ScID, SnID). We show in §3.2, an implementation of the above model for process-level, cluster wide access control mechanisms. A more detailed article on the implementation can be found in [2].

### 2.2 Mandatory Access Control at cluster level

In *Discretionary Access Control* (DAC), the objects' permissions are set by their owners. So, as soon as gets hold of a (buggy) process, he gains access to *all resources* available to the owner. This is used in several buffer overflow exploits to allow attacker to gain root privilege.

Adding the *Mandatory Access Control* to Linux remedies this problem. In MAC, access control no longer solely depends on the user's decision but also on a variety of security-relevant information. For example, executing a given process requires the correct Unix permissions, but also that the current security context permits the creation of a new process. This is particularly useful when the administrator needs to execute two types of program: `secure-prog` that s/he trusts and `handle-with-care-prog` that s/he does not entirely trust. Therefore, s/he considers that `secure-prog` should be allowed to spawn new processes, but `handle-with-care` should not. This is what MAC provides: it clearly assigns different security contexts to these two programs. With the DAC mechanisms only implemented, the administrator needed to create separate groups/users for both programs and avoid shared resources between two groups/users, and set the strict permissions for each groups/users. This is clear that this is not sustainable effort for an important number of binaries.

Even though the Linux community has not yet come up with a standard way of implementing MAC mechanisms inside Kernel Linux, several projects exist (c.f. SE Linux [5]...) implementing MAC approach at Linux kernel level. However, those solutions are still dedicated to single nodes; The DSI project allowed us to prove the feasibility of extending the MAC mechanisms to the distributed systems.

With persistent ScIDs for processes across the cluster, we extend the access control checks at kernel level from local nodes into the entire cluster. The ScID and SnID of the process initiating a connection are carried with the IP packet to the other end of the connection. This way, the permissions of the processes to access resources are verified in the entire cluster independently from their location. Note that these verifications are at kernel level, and are independent from the mechanisms to be implemented at application layer by the developers (this is particularly important when running untrusted code, or software that can not be modified with security considerations for technical or historical issues). We detail the above model in §3.2.

With MAC at the cluster level supporting process-level granularity, we have the necessary means to implement security zones inside the cluster.

### 2.3 Security zones

In order to compartmentalize the system, we define different security zones within the cluster and enforce the security policy for different security zones (see Figure 1).

The security zones are created by defining the security rules for interactions between different processes and resources through the cluster. For example, it is possible to define an access control rule stating that processes on node 1, with ScID of 2 may create sockets and connect to the processes of node 2, with ScID 2.

That is by assigning the same ScIDs to the processes and

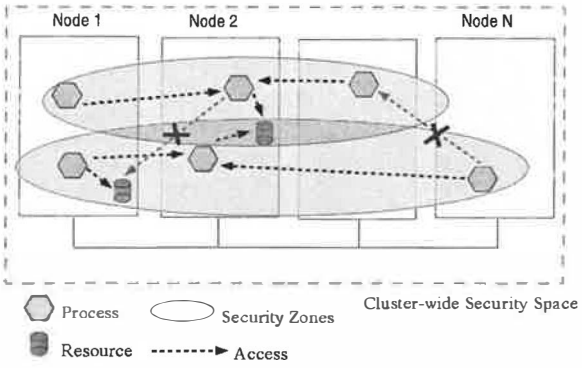



Figure 1: *Security zones are defined in the logical cluster-wide security space.*

resources of the same security zone, and defining rules to restrict any connection or access to resources for the zone defined by the ScID, the administrator can define a security zone throughout the cluster independently to where the processes are running or where the resources are located in the cluster. Note that the administrator can create a zone with privileges to access all zones for administrative purposes, or define the shared resources among different zones.

All security rules are collected in the *distributed security policy* (DSP) in order to define a unique, homogeneous and cluster-wide security policy to be enforced all over the nodes of the cluster.

Hence, the DSP simply consists of a list of rules to be applied to pairs of (SnID, ScID). Through the DSP, security rules can be set for each (SnID,ScID) pair, thus enabling a fine-grained process-level security policy, valid across the entire cluster.

The DSP is automatically propagated to all nodes of the cluster at initialization time, and after each change updates are sent to all nodes of the cluster. The new rules are then locally compiled and cached in the kernel memory for the fast access (this is done by the security managers in each node within the cluster, see §3).

Note that defining security zones is then simplified to editing DSP to set the same ScID for all processes and resources belonging to the same security zone (see details in §3.2).

# 3 An implementation of the model: DSI

To validate our approach and show its feasibility, an open source project, named *Distributed Security Infrastructure* (DSI), was initiated in 2002, so as to propose an adequate security solution for carrier-grade clusters. DSI implements the above mentioned security model [1].

DSI is composed of one security server (SS) and multiple security managers (SMs) - one per node. The SS is the central point of management of the cluster: it gathers all alarms and warnings sent by the SMs and propagates the security policy over the cluster. Each SM is responsible to enforce security on its own node.

DSI is based on open and standard software such as Linux Security Modules (LSM) for kernel level security mechanisms, OmniORB, an open-source implementation of Corba [3] and SSL/TLS for communication security.

Administrative messages between SMs and SS are sent on secure encrypted and authenticated channels, using SSL/TLS (i.e.; Secure Communication Channel in Figure 2).

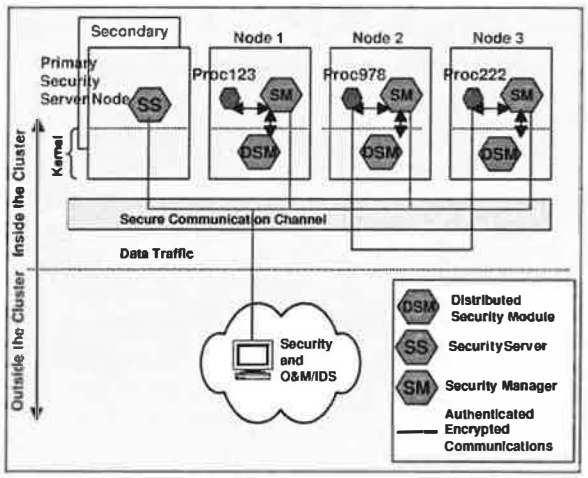



Figure 2: *Distributed Architecture of DSI*

## 3.1 The Distributed Security Policy (DSP)

In DSI, all security rules are defined through the Distributed Security Policy (DSP). The goal of the DSP is to define a unique, homogeneous and cluster-wide security policy to be enforced over all nodes of the cluster. It

```
<SOCKET_class_rule>
  <sSnID> 1 </sSnID>
  <sScID> 2 </sScID>
  <tSnID> 2 </tSnID>
  <tScID> 2 </tScID>
  <allow> CREATE  </allow>
</SOCKET_class_rule>
```

Figure 3: *Access control rule example for a socket.*

contains customization for all security services running on DSI.

Basically, an access control rule consists in various permissions to be applied to entities (i.e.; processes, sockets,...) sharing the same security context and security node identifier (i.e.; a ScID/SnID pair). Permissions are organized in different *classes*. For example, there are permissions relative to sockets (create, bind, send, receive), others relative to process creation. All kinds of permissions have not been implemented yet. Actually, we have mainly focused on network communication and process creation so far.

Currently, the DSP supports the following rule types:

- Process class allowing or denying a given pair (ScID, SnID) permission to spawn new processes. The DSP enables control over `fork()` or `execve()` system calls (c.f. §3.2.2).
- Socket class controlling specifically permissions of sockets on the cluster. ScIDs may be assigned to sockets of a given node, for a given protocol and port (currently only TCP and UDP are supported.). Then, it is possible to set permissions between source and target sockets/processes.
- Networking class controlling network permissions on the cluster, such as allowing or denying a given pair (ScID, SnID) to receive network information from a given pair (ScID, SnID).
- Transition class defining how ScIDs are assigned to processes according to the ScID stored in their binary file and their parent process.

For instance, Figure 3.1 illustrates an access control rule. It states that resources on node 1, with ScID of 2 may create a sockets to the processes in node 2, with ScID 2. All other connections from a process with ScID 2 in node 1 to any process in node 2 will be denied.

In order to be flexible and to ease human readability of configuration file, the DSP was implemented using XML language. On top of those extensibility features, XML comes with a variety of open source tools and with its own security mechanisms [4].

In the following, we do not explain DSI in details but rather concentrate on a practical example to illustrate our approach: the distributed access control mechanisms. Further details on DSI and DSP can be found in [1].

### 3.2 A practical case: the DIstributed Security Access Control service (DisAC)

DisAC implements the MAC paradigm over the *entire* cluster with process-level granularity as this was discussed in §2.2.

DSP allows to maintain a homogeneous, central point of security management for the cluster. The security administrator sets up the security policy on the security server, and then, the DSP gets propagated across the cluster.

DisAC also allows administrators to simplify access control rules by setting different categories of security contexts.

In the following we detail main functionality of the system. A detailed explanation of DisAC can be found in [2].

#### 3.2.1 Cluster-wide access control for DisAC

DisAC extends the local access control to a distributed access control for the whole cluster, using both source/target security node and security context identifiers as security information: $AccessDecision = Func(Source < SnID, ScID >, Target < SnID, ScID >)$.

The access decisions are enforced locally by DSM based on DSP rules for local and remote processes accessing local resources. For example in Figure 4, we illustrates the case where a process accesses a remote resource. First, a local check is performed to verify the access permissions of the local entity (here, process 12, ScID=10) to use network resources (DisAC by default assigns the ScID of the process creating the socket to the socket, therefore the local TCP socket ScID is 10). If permission is granted, ScID and SnID of the source entity (here process 12, ScID=10) are added to all IP packets sent from this socket. ScID and SnID are added in the IP Options

based on FIPS definition of standard security labels for information transfer [7]. When the IP packet is received on the remote node, the DSM module extracts the ScID and SnID from IP Packet and checks the access permission to the defined resource (here port 8000, ScID=10, DisAC allows to set ScIDs explicitly for each TCP or UDP port). Furthermore, a check is performed to verify that source entity has the permission to send information to the parent process which created the socket (here this comes to check permissions between process 12 and process 14). Finally, of course, DSM verifies that the process 14 has permission to receive information from TCP socket port 8000.

Therefore, access privileges may be defined at process-level for both local and remote nodes. For example, it is possible to define that a process of type A is only able to access resources of type B on nodes $M$ and $N$ of a given cluster.

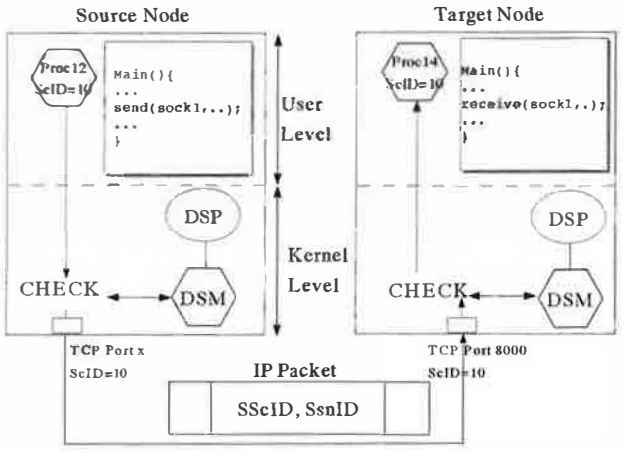



Figure 4: *Secure remote access control.*

### 3.2.2 Storing security context information (ScID) into binaries

A newly created process is automatically assigned a ScID by the DSM (see §3.2.3) based on the loaded binary file , the parent process and the security context of the system.

In DSI, we store a ScID in each binary (stored in its Elf header). This allows dividing binaries into different types based on the information for the provider of the software, or the level of trust for the binary. DSI also has mechanisms to support digital signatures for binaries in order to avoid these ScIDs to be tampered with by the intruders.

DisAC is particularly useful for large clusters, where there is a need for compartmentalization into distinct sub-clusters with restricted/controlled connections between sub-clusters. For instance, this scenario is quite useful for telecommunication clusters that are shared among different operators: operators share the global infrastructure of the cluster providing different services, but they certainly do not wish to share their binaries or data with other operators.

Therefore, the operator assigns the same ScID for the binaries provided by each operator and the resources to be accessed on different nodes by that application. Furthermore, the operator defines the access rules among the ScIDs, setting up possible interactions between different security zones.

### 3.2.3 Access control at kernel level

For security not to be bypassed, the DisAC service has been implemented at kernel level, in the *Distributed Security Module* (DSM). DSM is a set of kernel functions enforcing distributed security policy, and is implemented using Linux Security Module hooks as a Linux kernel module [6].

## 3.3 DSI Benchmark results

Tests have been performed using LMBench 3.0 [8], on two different configurations: a 2.4.17 kernel with the LSM [6] patch, without any security check performed and the same patched kernel, with the DSM module loaded, implementing different security mechanisms defined in DisAC (see Table 1).

UDP and TCP latency tests are performed by having client and server loop on exchanging a message of 4 bytes. The RPC tests are similar, but using Sun's RPC layer over TCP or UDP (see [8]).

DSI involves minimal overhead for local operations. For TCP and UDP tests, DSM's overhead ranges from 4% to 15% as security checks have to be done before processes are allowed to communicate and at the reception of each IP packet. The TCP Connect result is particularly heavy for DSM because TCP is a "three-way handshake" and security checks are done at each stage of the handshake. For RPC tests, as the overhead due to RPC connections increases in the total time of communication, the over-

| Test type | Without DSM | With DSM | Overhead |
|---|---|---|---|
| Stat | 1.92 | 1.94 | 1.0% |
| Open / Close | 2.68 | 2.68 | 0% |
| Fork | 92.81 | 93.58 | 0.82% |
| Exec | 322.56 | 328.33 | 1.78% |
| Sh proc | 2140.75 | 2150 | 0.43% |
| UDP | 9.68 | 10.61 | 9.6% |
| RPC/UDP | 17.66 | 18.7 | 5.9% |
| TCP | 11.08 | 12.68 | 14.4% |
| RPC/TCP | 23.42 | 24.3 | 3.75% |

Table 1: *Comparison of performances between a LSM patched kernel without any security mechanisms implemented and a kernel supporting DSI distributed security services. Tests have been done on an Intel Pentium IV 2.4 GHz. Time units are microseconds.*

head due to security checks decreases in percentage.

## 4 Future work

During the work with DSI, we realized that having implemented many of necessary mechanisms, one of the major challenges is creating the "right" distributed security policy (DSP). For time being, all interactions between different pairs (ScID, SnID) must be explicitly defined in the DSP. This task when many applications are involved becomes quite complex. We plan to simplify the DSP creation and sub-sequent modifications by generating the rules to allow interactions between all entities (processes and resources) of the same security zone. We avoid implementing this as a default rule inside DSP as we believe this could be too restricting in some cases. Still, one major issue would be how to handle interactions between different security zones in a user friendly way.

The security for communications between nodes is guaranteed using IPSec. We plan to implement a more fine grained security mechanism based on security policy set for each ScID. This comes to define security mechanisms for communications between processes inside a security zone independently from the location of processes within the cluster.

## 5 Conclusion

In this paper, we proposed a new security model for Linux clusters based on security zones allowing a flexible compartmentalization of software components comprising large distributed applications. This security model is based on defining process-level security rules and expanding MAC mechanisms to the whole cluster to enforce those rules.

## References


[1] *The Distributed Security Infrastructure Open Source Project*, `http://disec.sourceforge.net`.

[2] M. Pouzandi, A. Apvrille, E. Gingras, A. Medenou, D. Gordon, *Distributed Access Control for Carrier Class Clusters*, in the Proceedings of the Parallel and Distributed Processing Techniques and Applications (PDPTA'03), Las Vegas, June 2003.

[3] *omniORB*, `http://omniORB.sourceforge.net/`

[4] Eastlake D., Reagle J., Solo D., *XML-Signature Syntax & Processing*, Network Working Group, RFC 3275, March 2002.

[5] P. Loscocco, S. Smalley, *Integrating Flexible Support for Security Policies in the Linux Operating System*, in the Proceedings of the FREENIX track of the 2001 USENIX Annual Technical Conference, 2001, `http://www.nsa.gov/selinux`.

[6] C. Wright, C. Cowan, S. Smalley, J. Morris, G. Kroah-Hartmann, *Linux Security Modules: General Security Support for the Linux Kernel*, in the Proceedings of the 2002 USENIX Security Symposium, `http://lsm.immunix.org`.

[7] J. Morris, *SelOpt: Labeled IPv4 networking for SE Linux*, `http://www.intercode.com.au/jmorris/selopt`.

[8] *LmBench:* `http://www.bitmover.com/lmbench`.

# THE USENIX ASSOCIATION

Since 1975, the USENIX Association has brought together the community of developers, programmers, system administrators, and architects working on the cutting edge of the computing world. USENIX conferences have become the essential meeting grounds for the presentation and discussion of the most advanced information on new developments in all aspects of advanced computing systems. USENIX and its members are dedicated to:

- problem-solving with a practical bias
- fostering innovation and research that works
- communicating rapidly the results of both research and innovation
- providing a neutral forum for the exercise of critical thought and the airing of technical issues

### Member Benefits

- Free subscription to *;login:*, the Association's magazine, published six times a year, featuring technical articles, system administration tips and techniques, practical columns on Perl, Java, Tcl/Tk, and Open Source, book and software reviews, summaries of sessions at USENIX conferences, and Standards Reports from the USENIX representative and others on various ANSI, IEEE, and ISO standards efforts.
- Access to *;login:* on the USENIX Web site.
- Access to papers from the USENIX Conferences and Symposia, starting with 1993, on the USENIX Web site.
- Discounts on registration fees for the annual, multi-topic technical conference, the System Administration Conference (LISA), and the various single-topic symposia addressing topics such as security, Linux, Internet technologies and systems, operating systems, and Windows—as many as twelve technical meetings every year.
- Discounts on the purchase of proceedings and CD-ROMs from USENIX conferences and symposia and other technical publications.
- The right to vote on matters affecting the Association, its bylaws, and election of its directors and officers.
- Savings on a variety of products, books, software, and periodicals: see *http://www.usenix.org/membership/specialdisc.html* for details.

# SAGE

SAGE is a Special Technical Group (STG) of the USENIX Association. It is organized to advance the status of computer system administration as a profession, establish standards of professional excellence and recognize those who attain them, develop guidelines for improving the technical and managerial capabilities of members of the profession, and promote activities that advance the state of the art or the community.

## USENIX & SAGE Thank Their Supporting Members

### USENIX Supporting Members

❖ Addison-Wesley/Prentice Hall PTR ❖ Ajava Systems, Inc. ❖

❖ Aptitune Corporation ❖ Atos Origin B.V. ❖ Delmar Learning ❖

❖ DoCoMo Communications Laboratories USA, Inc. ❖

❖ Electronic Frontier Foundation ❖ Hewlett-Packard ❖ Interhack Corporation ❖

❖ MacConnection ❖ The Measurement Factory ❖ Microsoft Research ❖ Portlock Software ❖

❖ Raytheon ❖ Sun Microsystems, Inc. ❖ Taos Mountain, Inc. ❖

❖ UUNET Technologies, Inc. ❖ Veritas Software ❖

### SAGE Supporting Members

❖ Addison-Wesley/Prentice Hall PTR ❖ Ajava Systems, Inc. ❖

❖ Microsoft Research ❖ MSB Associates ❖ Raytheon ❖ Ripe NCC ❖ Taos Mountain, Inc. ❖

For more information about membership, conferences, or publications,
see *http://www.usenix.org/*
or contact:

USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710 USA
Phone: 510-528-8649 Fax: 510-548-5738 Email: *office@usenix.org*

ISBN 1-931971-22-6